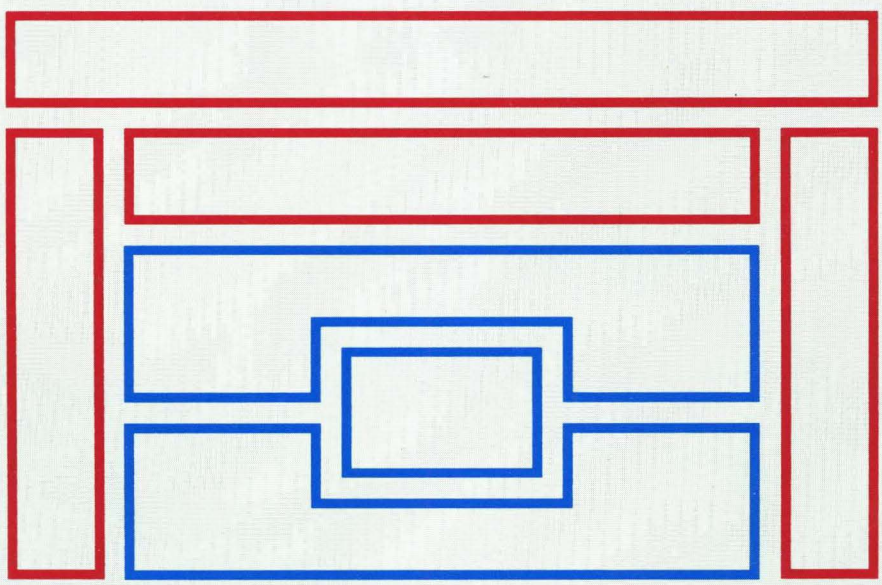
IBM

Systems Application Architecture

SC26-4399-1

Common Programming Interface Communications Reference



Systems Application Architecture

SC26-4399-1

Common Programming Interface Communications Reference

Second Edition (October 1988)

This edition applies to IBM's Systems Application Architecture as announced in March 1987, and expanded in October of that year.

Summary of Changes

The major change of this book over the SC26-4399-0 version is the inclusion of Appendixes E and F, which provide, respectively, VM-specific information and example COBOL programs. Minor changes to the text and illustrations have been indicated by a vertical line to the left of the change. Editorial changes that have no technical significance are not noted.

Before using this publication in connection with the operation of IBM systems, consult the latest *IBM System/370, 30xx, and 4300 Processors Bibliography*, GC20-0001, or *IBM AS/400 Information Directory*, GC21-9678, for the editions that are applicable and current.

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.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not of itself constitute or imply a grant of (i) any license under any patents, patent applications, trademarks, copyrights, or other similar rights of IBM or of any third party; or (ii) any right to refer to IBM in any advertising or other promotional or marketing activities. IBM assumes no responsibility for any infringement of patents or other rights that may result from use of the subject matter described in this document or for the manufacture, use, lease, or sale of machines or programs described herein, outside of the responsibilities assumed via the agreement for purchase of IBM machines and the agreement for licensed programs.

Licenses under IBM's utility patents are available on reasonable and nondiscriminatory terms and conditions. IBM does not grant licenses under its appearance design patents. Inquiries relative to licensing should be directed in writing to the IBM Director of Commercial Relations, International Business Machines Corporation, Armonk, New York, 10504.

The following sentence does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: International Business Machines provides this publication "As Is" without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability or fitness for a particular purpose. Within the United States, some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality. If you request publications from the address given below, your order will be delayed because publications are not stocked there.

A form for readers' comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Information Development, Dept G60, P.O. Box 6, Endicott, NY, U.S.A., 13760. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Contents

Figures

Tables

Chapter 1. Introduction

This introductory chapter:

- Identifies the book's purpose, structure, and audience
- Gives a brief overview of Systems Application Architecture™ (SAA)
- Explains how to use the book.

Purpose and Structure

This book defines the Communications element of SAA's Common Programming Interface (CPI). CPI Communications provides a programming interface that allows program-to-program communications using IBM's Systems Network Architecture (SNA) logical unit 6.2 (LU 6.2). In addition to this chapter, this book contains the following sections:

- **Chapter 2, "CPI Communications Terms and Concepts"**

  This chapter describes basic terms and concepts used in CPI Communications.

- **Chapter 3, "Program-to-Program Communication Tutorial"**

  This chapter provides a number of sample flows that show how a program can combine CPI-Communications calls for program-to-program communication.

- **Chapter 4, "Reference Section"**

  This chapter describes format and function of each of the CPI-Communications calls.

- **Appendix A, "Variables and Characteristics"**

  This appendix describes the CPI-Communications variables and conversation characteristics.

- **Appendix B, "Return Codes"**

  This appendix describes the return codes that may be returned when CPI-Communications calls are executed.

- **Appendix C, "State Table"**

  This appendix explains when and where the CPI-Communications calls can be issued.

- **Appendix D, "CPI Communications and LU 6.2"**

  For programmers who are familiar with the LU 6.2 application programming interface, this appendix explains the relationship between LU 6.2 verbs and CPI-Communications calls.

- **Appendix E, "CMS VM/SP—Extension Information"**

  This appendix contains information about VM/SP extensions to CPI Communications.

Systems Application Architecture is a trademark of International Business Machines Corporation.

- **Appendix F, "Sample Programs"**

  This appendix contains two sample COBOL programs using CPI Communications.

Who Should Read This Book

This book is intended for programmers who want to write applications that adhere to the Communications interface. Although the book is designed as a reference, Chapter 3, "Program-to-Program Communication Tutorial" provides a tutorial on designing application programs using CPI-Communications concepts and calls.

What Is Systems Application Architecture

SAA is a definition — a set of software interfaces, conventions, and protocols that provide a framework for designing and developing applications with cross-system consistency.

Systems Application Architecture:

- Defines a **common programming interface** that can be used to design applications easily integrated with other applications and made to run in multiple SAA environments
- Defines **common communications support** for connection of applications, systems, networks, and devices
- Defines a **common user access** that allows consistency in panel layout and user interaction techniques.
- Offers some **common applications** written by IBM using the above.

Supported Environments

SAA provides a framework across these IBM computing environments:

- TSO/E in the Enterprise Systems Architecture/370™
- CMS in the VM/System Product or VM/Extended Architecture
- Operating System/400™(OS/400™)
- Operating System/2™(OS/2™) Extended Edition
- IMS/VS Data Communications in the Enterprise Systems Architecture/370™
- CICS/MVS in the Enterprise Systems Architecture/370™ .

Operating System/2, Operating System/400, Enterprise Systems Architecture/370, OS/2, and OS/400 are trademarks of the International Business Machines Corporation.

Common Programming Interface

As its name implies, the CPI provides languages, commands, and calls that allow development of applications that will take advantage of the consistency offered by SAA. These applications are then more easily integrated and transported across the supported environments.

The components of the interface currently fall into two general categories:

- Languages

  Application Generator
  C
  COBOL
  FORTRAN
  Procedures Language
  RPG

- Services

  Communications Interface (defined by this book)
  Database Interface
  Dialog Interface
  Presentation Interface
  Query Interface

The CPI is defined by this manual and the other CPI reference books. It is not, in itself, a product or a piece of code. But — as a definition — it does establish and control how IBM products are being implemented, as well as providing a common base across the SAA environments.

In addition to the CPI, the other elements of SAA should be considered when designing an application. A list of SAA books can be found under "Related Publications" on page 5 and on the back cover of this book.

How to Use This Book

This section describes the relationship between SAA CPI Communications and the operating systems on which it is implemented, explains how that relationship is indicated in this book, and provides a general path through the book for a more comprehensive understanding of SAA CPI Communications.

Relationship to Products

SAA CPI Communications defines the elements that are consistent across the SAA environments. Preparing and running programs requires the use of a CPI-Communications product that implements SAA on one of those systems. CMS is the current implementing product for CPI Communications.

CMS has its own set of product documentation, which will be required in addition to this one. The product documentation describes additional system-dependent elements, such as, for example, how to prepare and run a program in that particular environment.

See "Related Publications" on page 5 for a list of the product documentation currently available.

How Product Implementations Are Designated

Because SAA is still evolving, complete and consistent products may not be available yet on all systems. Some interface elements may not be implemented everywhere. Others may be implemented, but differ slightly in their syntax or semantics (how they are coded or how they behave at run time).

These conditions are identified in this book in two ways:

- A system checklist precedes each interface element. If the interface element is implemented on a particular system, that column is marked with an X. If it is not yet implemented on a particular system, that column is blank. In the following example, an implementation is available on CMS, but not on TSO/E, OS/400, OS/2, IMS, or CICS.

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

- The SAA Communications Interface definition is printed in black ink. If the implementation of an interface element in an operating environment differs in its syntax or semantics, the text states that fact and is printed in green — as is this sentence.

Syntax

A discussion of the conventions used specifically by CPI Communications within this manual can be found in "Naming Conventions — Calls and Characteristics, Variables and Values" on page 19.

General Path through the Manual

This book contains both tutorial and reference information. Use the path identified below to achieve the most benefit.

- Read Chapter 2, "CPI Communications Terms and Concepts" for an overview of the terms and concepts used in CPI Communications. It is required to understand the sample program flows shown in Chapter 3, "Program-to-Program Communication Tutorial."
- Chapter 3, "Program-to-Program Communication Tutorial" explains how to use the CPI-Communications calls and provides examples. When reading this chapter, use Chapter 4, "Reference Section" to obtain additional information about the function of and required parameters for the CPI-Communications calls.
- Use Chapter 4, "Reference Section" and the appendixes for specific functional information on how to code applications.

Related Publications

The following manuals provide additional information on SAA and the operating systems on which it is implemented.

For Systems Application Architecture

CPI reference manuals describe each component of the common programming interface:

Application Generator Reference (SC26-4355)
C Reference (SC26-4353)
COBOL Reference (SC26-4354)
Communications Reference (SC26-4399)
Database Reference (SC26-4348)
Dialog Reference (SC26-4356)
FORTRAN Reference (SC26-4357)
Presentation Reference (SC26-4359)
Procedures Language Reference (SC26-4358)
Query Reference (SC26-4349).

The following publications may also prove useful:

Systems Application Architecture: An Overview (GC26-4341)

Introduces SAA concepts, and identifies the environments and elements that participate.

Common User Access: Panel Design and User Interaction (SC26-4351)

Defines the common user access for Personal Computers and System/370 and AS/400™ terminals, including panel layout and user interaction techniques.

Writing Applications: A Design Guide (SC26-4362)

Provides guidance on developing application programs that are consistent and portable across the SAA environments. These applications will use the common programming interfaces and implement the common user access specification.

For Implementing Products

- *VM/SP Connectivity Programming Guide and Reference*, SC24-5337
- *VM/SP Connectivity, Planning, Administration, and Operation*, SC24-5338.

For LU 6.2

- *SNA Formats, GA27-3136*
- *Systems Network Architecture Concepts and Products*, GC30-3072
- *Systems Network Architecture Technical Overview*, GC30-3073.
- *SNA Transaction Programmer's Reference Manual for LU Type 6.2, GC30-3084*
- *SNA LU 6.2 Reference: Peer Protocols, SC31-6808*

AS/400 is a trademark of the International Business Machines Corporation.

Interface Definition Table

Table 1 lists the calls currently defined in the communications interface for SAA. An X is used to indicate which systems already have an IBM licensed program announced or available that implements a particular communications call.

Table 1. Major Elements of the Communications Interface

Call Name	TSO/E	CMS	OS/400	OS/2	IMS	CICS
Starter Set						
Accept_Conversation		X				
Allocate		X				
Deallocate		X				
Initialize_Conversation		X				
Receive		X				
Send_Data		X				
Advanced Function						
for synchronization and control:						
Confirm		X				
Confirmed		X				
Flush		X				
Prepare_To_Receive		X				
Request_To_Send		X				
Send_Error		X				
Test_Request_To_Send_Received		X				
for modifying conversation characteristics:						
Set_Conversation_Type		X				
Set_Deallocate_Type		X				
Set_Error_Direction		X				
Set_Fill		X				
Set_Log_Data		X				
Set_Mode_Name		X				
Set_Partner_LU_Name		X				
Set_Prepare_To_Receive_Type		X				
Set_Receive_Type		X				
Set_Return_Control		X				
Set_Send_Type		X				
Set_Sync_Level		X				
Set_TP_Name		X				
for examining conversation characteristics:						
Extract_Conversation_Type		X				
Extract_Mode_Name		X				
Extract_Partner_LU_Name		X				
Extract_Sync_Level		X				

Chapter 2. CPI Communications Terms and Concepts

CPI Communications provides a consistent application programming interface for applications that require program-to-program communication. The interface makes use of SNA's LU 6.2 to create a rich set of inter-program services, including:

- Sending and receiving data
- Synchronizing processing between programs
- Notifying a partner of errors in the communication.

This chapter describes the major terms and concepts used in CPI Communications.

Communication across an SNA Network

Figure 1 illustrates the logical view of an example SNA network. It consists of three logical units (LUs): LU X, LU Y, and LU Z. Each LU is involved in two sessions (the gray portions of Figure 1). A **session** is the logical connection between two LUs. The network shown in Figure 1 is a simple one. In a real network, the number of LUs can be in the thousands.

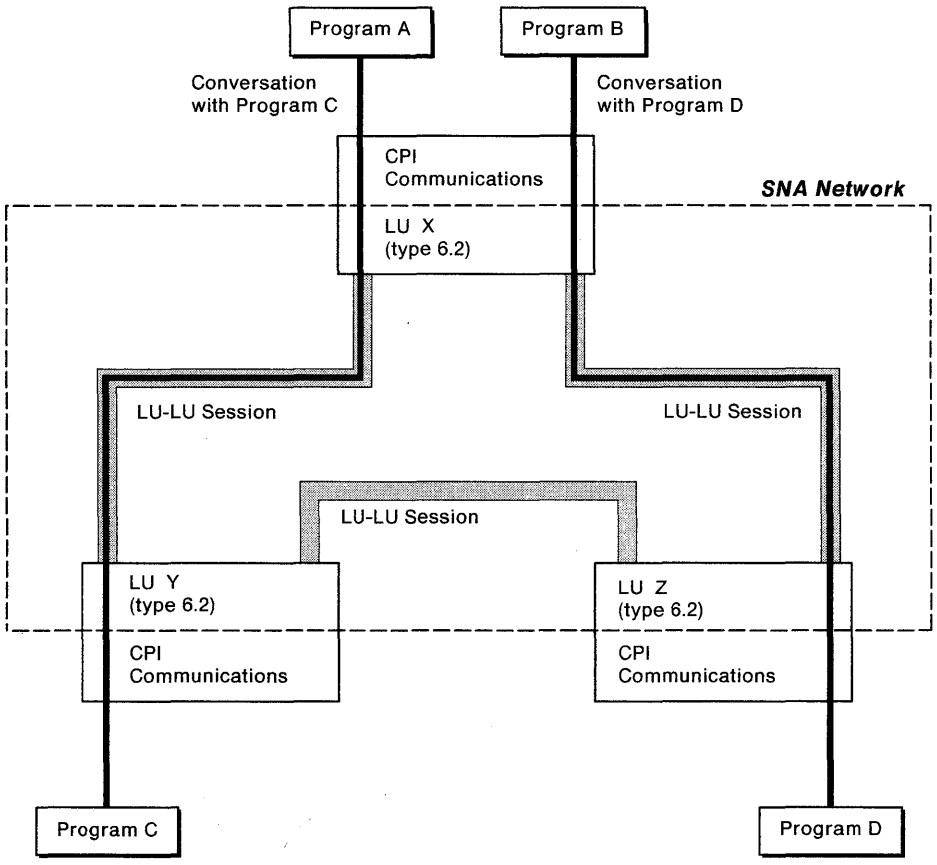


Figure 1. Programs Using CPI Communications to Converse Through an SNA Network

The LUs and the sessions shown in Figure 1 are of type 6.2. Although SNA defines many types of LUs, CPI Communications only uses LU 6.2 sessions.

Note: The physical network, which consists of nodes (processors) and data links between nodes, is not shown in Figure 1 because a program using CPI Communications does not "see" these resources. A CPI-Communications program uses the logical network of LUs, which in turn communicates with, and uses, the physical network. For more information on SNA networks, refer to *Systems Network Architecture Concepts and Products* and *Systems Network Architecture Technical Overview*.

Program Partners and Conversations

Just as two LUs communicate using LU-6.2 sessions, two CPI-Communications programs exchange data using a **conversation**. For example, the conversation between Program A and Program C is shown in Figure 1 as a single bold line between the two programs. The line indicating the conversation is shown on top of the session because a conversation connects programs "over" a session.

CPI Communications supports two types of conversations:

- **Mapped** conversations allow programs to exchange arbitrary **data records** in data formats agreed upon by the application programmers.
- **Basic** conversations allow programs to exchange data in a standardized format. This format is a stream of data containing 2-byte length fields (referred to as LLs) that specify the amount of data to follow before the next length field. The typical data pattern is "LL, data, LL, data." Each grouping of "LL, data" is referred to as a **logical record**.

  Note: Because of the detailed manipulation of data and resulting complexity of error conditions, the use of basic conversations should be regarded as intended for advanced programmers. A more complete discussion of basic and mapped conversations is provided in the "Usage Notes" section of "Send_Data (CMSEND)" on page 83.

For further information on basic and mapped conversations, refer to *SNA LU 6.2 Reference: Peer Protocols* and *SNA Transaction Programmer's Reference Manual for LU Type 6.2*.

Two programs involved in a conversation are called **partners** in the conversation. If an LU-LU session exists, or can be made to exist, between the nodes containing the partner programs, two CPI-Communications programs can communicate through an SNA network over a conversation.

The terms local and remote are used to differentiate between different ends of a conversation. If a program is being discussed as **local**, its partner program is said to be the **remote** program for that conversation. For example, if Program A is being discussed, Program A is the local program and Program C is the remote program. Similarly, if Program C is being discussed as the local program, Program A is the remote program. Thus, a program can be both local and remote for a given conversation, depending on the context.

Operating Environment

Figure 2 provides a more detailed view of Program A's operating environment, focusing on the node of the network that contains Program A. Lines showing services and connections to different generic components of the node indicate the components used by Program A to establish communication with another program.

In Figure 2, there are two lines connected to the bottom of the Program A block. The line on the right indicates the conversation as previously shown in Figure 1 on page 8. The new line on the left shows Program A's communication with the communications element itself. This line represents a specific set of program calls that can be made using the local system's library of common code. The different types of CPI-Communications calls that a program may make are discussed later in this chapter in "Program Calls."

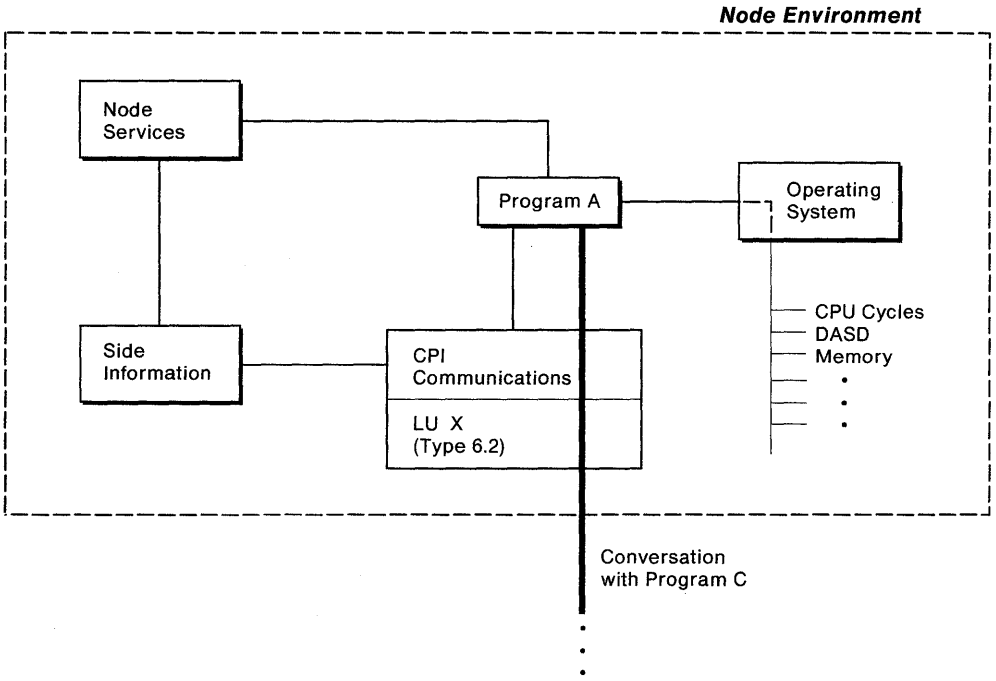


Figure 2. Operating Environment of CPI-Communications Program

In addition to the new line with CPI Communications, Figure 2 shows three new generic elements in communication with Program A:

- Side information
- Node services
- Operating system.

These new elements are discussed in the following sections.

Side Information

For a program to establish a conversation with a partner program, CPI Communications requires a certain amount of initialization information, such as the name of the partner program and the name of the LU at the partner's node. CPI Communications provides a way to use system-defined values for these required fields; these system-defined values are called **side information**.

System administrators supply and maintain the side information for all CPI-Communications programs. The side information is accessed by a symbolic destination name. The **symbolic destination name**, referred to as *sym_dest_name* in this book, corresponds to an entry in the side information containing the following three pieces of information:

- ***partner_LU_name***

  Indicates the name of the LU where the partner program is located. This LU name is any name for the remote LU recognized by the local LU for the purpose of allocating a conversation.

- ***mode_name***

  Used by LU 6.2 to designate the properties for the session that will be allocated for the conversation. The properties include, for example, the class of service to be used on the conversation.

  The system administrator defines a set of mode names for each partner LU with which the local LU communicates. This set of mode names can differ from one partner LU to another.

- ***TP_name***

  Specifies the name of the remote program. See Appendix D, "CPI Communications and LU 6.2" on page 147 for details of how a CPI-Communications program can interact with non-CPI-Communications programs.

  Note: *TP_name* stands for "transaction program name," which comes from the LU 6.2 application programming interface where the programs are referred to as transaction programs. In this manual, transaction program, application program, and program are synonymous, all denoting a program using CPI Communications.

Node Services

Node services represents a number of "utility" type functions within the local system environment that are available for CPI Communications, as well as other elements of the CPI. These functions are not related to the actual sending and receiving of CPI-Communications data, and specific implementations differ from product to product. Node services include the following general functions:

- Setting and accessing of side information by system administrators

  This function is required to set up the initial values of the side information and allow subsequent modification. It does not refer to individual program modification of the program's copy of the side information using Set calls as described in "Conversation Characteristics" on page 14. (Refer to specific product information for details.)

- Program-start-up processing

  A program is started either by receipt of notification that the remote program has issued an Allocate call for the conversation (discussed in greater detail in "Starter-Set Flows" on page 22) or by local (operator) action. In either case, node services sets up the required data paths and operating environment required by the program and then allows the program to begin execution. In the former case, node services receives the notification, retrieves the name of the program to be started, and then proceeds as if starting a program by local action.

- Program-termination processing (both normal and abnormal)

  The program should terminate all conversations before the end of the program. However, if the program does not terminate all conversations, node services is responsible for deallocating any dangling conversations.

  On VM, facilities such as nucleus extensions and discontiguous saved segments, which permit programs to be memory resident and "runnable" even after returning control to CMS, prevent CMS from determining that a conversation has been abandoned. Therefore, in a VM environment, conversations are immediately deallocated only in the case of an abnormal return to CP or CMS.

  If the program finishes normally and has not terminated all conversations, the conversations will be deallocated only if the user enters an HX command, resets the virtual machine (using IPL), or logs-off. If a program in a VM environment does not explicitly deallocate all active conversations, SNA network resources will be unavailable (for use by other programs) for longer periods of time than is required.

  See "Deallocate (CMDEAL)" on page 56 for more information on deallocating conversations.

Operating System

CPI Communications depends on the operating system for the normal execution and operation of the program. Activities such as linking, invoking, and compiling programs are all described in product documentation.

Program Calls

CPI Communications programs communicate with each other by making program **calls.** These calls are used to establish the **characteristics** of the conversation (conversation characteristics are discussed in greater detail in the next section, "Conversation Characteristics") and to exchange data and control information between the programs. The function provided by the CPI-Communications calls can be categorized into two groups:

- **Starter-Set Calls**

  The starter-set calls allow for simple communication of data between two programs and assume the program uses the initial values for the CPI Communications conversation characteristics. Example flows for use of these calls is provided in "Starter-Set Flows" on page 22.

- **Advanced-Function Calls**

  The advanced-function calls are used to do more specialized processing than that provided by the default set of characteristic values. The advanced-function calls provide more careful synchronization and monitoring of data. For example, the Set calls allow a program to modify conversation characteristics, and the Extract calls allow a program to examine the conversation characteristics that have been assigned to a given conversation. Example flows for use of these calls is provided in "Advanced-Function Flows" on page 29.

Note: The breakdown of function between starter-set and advanced-function calls is not intended to imply a restriction on how the calls may be combined or used. Starter-set calls, for example, will often be used together with advanced-function calls. The distinction between the two types of calls is intended solely as a documentation aid for the CPI-Communications programmer.

Table 2 on page 14 lists the two groups of CPI-Communications calls.

Table 2. Breakdown of Calls between Starter Set and Advanced Function

Starter Set	
Initialize_Conversation	
Accept_Conversation	
Allocate	
Send_Data	
Receive	
Deallocate	
Advanced Function	
Confirm	Set_Conversation_Type
Confirmed	Set_Deallocate_Type
Flush	Set_Error_Direction
Prepare_To_Receive	Set_Fill
Request_To_Send	Set_Log_Data
Send_Error	Set_Mode_Name
Test_Request_To_Send_Received	Set_Partner_LU_Name
	Set_Prepare_To_Receive_Type
	Set_Receive_Type
Extract_Conversation_Type	Set_Return_Control
Extract_Mode_Name	Set_Send_Type
Extract_Partner_LU_Name	Set_Sync_Level
Extract_Sync_Level	Set_TP_Name

A list of the calls and a brief description of each call's function is provided at the front of Chapter 4, "Reference Section" on page 44.

Conversation Characteristics

As discussed already in "Program Calls," CPI Communications maintains a set of characteristics for each conversation used by a program. These characteristics are established for each program on a per-conversation basis. The initial values assigned to the characteristics depend on the program's role in starting the conversation.

Here is a simple example of how Program A starts a conversation with Program C:

1. Program A issues the Initialize_Conversation call to start the conversation. It uses a *sym_dest_name* to designate Program C as its partner program and receives back a unique conversation identifier, the *conversation_ID*. Program A will use the *conversation_ID* in all future calls intended for that conversation.
2. Program A issues an Allocate call to start the conversation.
3. CPI Communications tells the node containing Program C that Program C needs to be started by sending a conversation start-up request to the partner LU.
4. Program C is started and issues the Accept_Conversation call. It receives back a unique *conversation_ID* (not necessarily the same as the one provided to Program A), which Program C will use in all future calls intended for that conversation.

After issuing their respective Initialize_Conversation and Accept_Conversation calls, both Program A and C have a set of default conversation characteristics set up for the conversation. Table 3 provides a comparison of the conversation characteristics and initial values as set by the Initialize_Conversation call (described on page 68) and the Accept_Conversation call (described on page 47). The values shown in the figure are pseudoynms that represent integer values.

The CPI Communications naming conventions for these characteristics, as well as for calls, variables, and characteristic values, are discussed in "Naming Conventions — Calls and Characteristics, Variables and Values" on page 19.

Table 3. Characteristics and Their Default Values

Name of Characteristic	Initialize_Conversation sets it to:	Accept_Conversation sets it to:
conversation_type	CM_MAPPED_CONVERSATION	The value received on the conversation start-up request
deallocate_type	CM_DEALLOCATE_SYNC_LEVEL	CM_DEALLOCATE_SYNC_LEVEL
error_direction	CM_RECEIVE_ERROR	CM_RECEIVE_ERROR
fill	CM_FILL_LL	CM_FILL_LL
log_data	Null	Null
log_data_length	0	0
mode_name	The mode name from side information referenced by *sym_dest_name*	The mode name for the session on which the conversation start-up request arrived
mode_name_length	The length of *mode_name*	The length of *mode_name*
partner_LU_name	The partner LU name from side information referenced by *sym_dest_name*	The partner LU name for the session on which the conversation start-up request arrived
partner_LU_name_length	The length of *partner_LU_name*	The length of *partner_LU_name*
prepare_to_receive_type	CM_PREP_TO_RECEIVE_SYNC_LEVEL	CM_PREP_TO_RECEIVE_SYNC_LEVEL
receive_type	CM_RECEIVE_AND_WAIT	CM_RECEIVE_AND_WAIT
return_control	CM_WHEN_SESSION_ALLOCATED	Null
send_type	CM_BUFFER_DATA	CM_BUFFER_DATA
sync_level	CM_NONE	The value received on the conversation start-up request
TP_name	The program name from side information referenced by *sym_dest_name*	Null
TP_name_length	The length of *TP_name*	0

Note: When the local program issues Initialize_Conversation, the default characteristics established include the three pieces of side information previously discussed: *partner_LU_name*, *mode_name*, and *TP_name*. For the remote program's characteristics, however, CPI Communications determines the *partner_LU_name* and *mode_name* from the session and conversation information provided by the LU in the conversation start-up request.

Modifying and Viewing Characteristics

In the previous example, the programs used the initial set of program characteristics provided by CPI Communications as defaults. However, CPI Communications provides calls that allow a program to modify and view the conversation characteristics for a particular conversation. Restrictions on when a program can issue one of these calls are discussed in the individual call descriptions in Chapter 4, "Reference Section."

Note: As already stated, CPI Communications maintains conversation characteristics on a per-conversation basis. Changes to a characteristic will affect only the conversation indicated by the *conversation_ID*. Changes made to a characteristic do not affect future default values assigned, nor do the changes (in the case of values derived from the side information) affect the initial system values.

For example, consider the conversation characteristic that defines what type of conversation the initiating program will have, the *conversation_type* characteristic. CPI Communications initially sets this characteristic to CM_MAPPED_CONVERSATION and stores this characteristic value for use in maintaining the conversation. A program can issue the Extract_Conversation_Type call to view this value.

A program can issue the Set_Conversation_Type call (after issuing Initialize_Conversation but before issuing Allocate) to change this value. The change remains in effect until the conversation ends or until the program issues another Set_Conversation_Type.

The Set calls are also used to prevent programs from attempting incorrect syntactic or semantic changes to conversation characteristics. For example, if a program attempts to change the *conversation_type* after the conversation has already been established (an illegal change), CPI Communications informs the program of its error and disallows the change. Details of this type of checking are provided in the individual call descriptions in Chapter 4, "Reference Section."

Program Flow — States and Transitions

As implied throughout the discussion so far, a program written to make use of CPI Communications is written with the remote program in mind. The local program issues a CPI-Communications call for a particular conversation with the knowledge that, in response, the remote program will issue another CPI-Communications call for that same conversation. To explain this two-sided programming scenario, CPI Communications uses the concept of a conversation state. The **state** that a conversation is in determines what the next set of actions may be. When a conversation leaves a state, it makes a **transition** from that state to another.

A CPI-Communications conversation can be in one of the following states:

State	Description
Reset	There is no conversation for this *conversation_ID*.
Initialize	Initialize_Conversation has completed successfully and a *conversation_ID* has been assigned for this conversation.
Send	The program is able to send data on this conversation.
Receive	The program is able to receive data on this conversation.
Send-Pending	The program has received both data and send capability on the same Receive call. See "Example 7: Error Direction and Send-Pending State" on page 38 for a discussion of **Send-Pending** state.
Confirm	A confirmation request has been received on this conversation; that is, the remote program issued a Confirm call and is waiting for the local program to issue Confirmed. After responding with Confirmed, the local program enters **Receive** state.
Confirm-Send	A confirmation request and permission-to-send have both been received on this conversation; that is, the remote program issued a Prepare_To_Receive call with the *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and the *sync_level* for this conversation is CM_CONFIRM. After responding with Confirmed, the local program enters **Send** state.
Confirm-Deallocate	A confirmation request and deallocation notification have both been received on this conversation; that is, the remote program issued a Deallocate call with the *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL and the *sync_level* for this conversation is CM_CONFIRM. After responding with Confirmed, the conversation is deallocated.

A conversation starts out in **Reset** state and progresses through the different states listed, depending on the calls made by the program for that conversation and the information received from the remote program. The current state of a conversation determines what calls the program can or cannot make.

Since there are two programs for each conversation (one at each end), the state of the conversation *as seen by each program* may be different. The state of the conversation depends on which end of the conversation is being discussed. Consider a conversation where Program A is sending data to Program C. Program A's end of the conversation is in **Send** state, but Program C's end is in **Receive** state.

Notes:

1. CPI Communications keeps track of a conversation's current state, as should the program. When the two are out of sync, a run-time error condition arises. This may be caused by an invalid program call or state transition. This error condition is indicated by a *return_code* value of CM_PROGRAM_STATE_CHECK.
2. When the meaning is unambiguous, this book sometimes refers to a program as being in a particular state. This means that one of the program's conversations (the one under discussion) is in a particular state.

For a complete listing of program calls and possible states and state transitions, see Appendix C, "State Table."

Naming Conventions — Calls and Characteristics, Variables and Values

Pseudonyms for the actual calls, characteristics, variables, states, and characteristic values that make up CPI Communications are used throughout this book to simplify understanding and readability. Where possible, underscores (_) and complete names are used in the pseudonyms. Any phrase in the book that contains an underscore is a pseudonym.

For example, Send_Data is the pseudonym for the program call, CMSEND, which is used by a program to send information to its conversation partner.

Note: The first two characters of all CPI Communications calls are a prefix of CM to aid in recognizing actual call names.

The book uses the following conventions to aid in distinguishing between the four types of pseudonyms:

- **Calls** begin with capital letters, as will each underscore-separated portion of the call's name. For example, **A**ccept_**C**onversation is the pseudonym for the actual call name of CMACCP.
- **Characteristics** and **variables** used to hold the values of characteristics are in italics (for example, *conversation_type*) and contain no capital letters except those used for abbreviations (for example, *TP_name*).

  In most cases, the parameter used on a call, which corresponds to a program variable, has the same name as the conversation characteristic. Whether a name refers to a parameter, a program variable, or a characteristic is determined by context. In all cases, the value used for the three remains the same.

  To indicate that a characteristic has been set to a particular value, the book will either say so explicitly or use "function" notation. For example, a *sync_level* of CM_NONE may also appear as *sync_level*(CM_NONE).
- **Values** used for characteristics and variables appear in all small uppercase letters (such as CM_OK) and represent actual integer values that will be placed into the variable. For a list of the integer values that are placed in the variables, see Table 4 on page 124 in Appendix A, "Variables and Characteristics."

- **States** are used to determine the next set of actions that can be taken in a conversation. States begin with capital letters and appear in bold type, such as, **Reset** state.

As a complete example of how pseudonyms are used in this book, suppose a program uses the Set_Return_Control call to set the conversation characteristic of *return_control* to a value of CM_IMMEDIATE.

- Chapter 4, "Reference Section" contains the syntax and semantics of the variables used for the call. It explains that the real name of the program call for Set_Return_Control is CMSRC (see "Set_Return_Control (CMSRC)" on page 112) and that CMSRC has a parameter list of *conversation_ID*, *return_control*, and *return_code*.
- Appendix A, "Variables and Characteristics" provides a complete description of all variables used in the book and shows that the *return_control* variable that goes into the call as a parameter is a 32-bit integer. This information is provided in Table 6 on page 129.
- Table 4 on page 124 in Appendix A, "Variables and Characteristics" shows that the value of CM_IMMEDIATE that is placed into the *return_control* parameter on the call to CMSRC is defined as an integer value of 1.
- Finally, the meaning of the *return_code* value CM_OK that is returned to the program on the call is provided in Appendix B, "Return Codes." It means that the call completed successfully.

Notes:

1. Pseudonym value names are not actually passed to CPI Communications as a string of characters. Instead, the pseudonyms represent the integer values that are passed on the program calls. The pseudonym value names are used to aid readability of the text. Similarly, programs should use translates and equates (depending on the language) to aid the readability of the code. In the above example, for instance, a program equate could be used to define CM_IMMEDIATE as meaning an integer value of 1. The actual program code would then read as described above, namely, that *return_control* is replaced with CM_IMMEDIATE. The end result, however, is that an integer value of 1 is placed into the variable.
2. "Programming Language Considerations" on page 177 in Appendix E, "CMS VM/SP—Extension Information" provides information on system files that can be used to establish pseudonyms for the program.

Chapter 3. Program-to-Program Communication Tutorial

This chapter provides example flows of how two programs using CPI Communications can exchange information and data in a controlled manner.

The examples are broken up into two sections:

- "Starter-Set Flows" on page 22
- "Advanced-Function Flows" on page 29.

In addition to these sample flows, a simple COBOL application using CPI-Communications calls is provided in Appendix F, "Sample Programs" on page 181.

A Word about the Flow Diagrams

In the flow diagrams shown in this chapter (for example Figure 3 on page 25), vertical dotted lines indicate the components involved in the exchange of information between systems. The horizontal arrows indicate the direction of the flow for that step. The numbers lined up on the left side of the flow are reference points to the flow and indicate the progression of the calls made on the conversation. These same numbers correspond to the numbers under the **Step** heading of the text description for each example.

The call parameter lists shown in the flows are not complete; only the parameters of particular interest to the flows being discussed are shown. A complete description of each CPI-Communications call and the required parameters can be found in Chapter 4, "Reference Section."

A complete discussion of all possible timing scenarios is beyond the scope of the chapter. Where appropriate, such discussion is provided in the individual call descriptions in Chapter 4, "Reference Section."

Note: When the meaning is unambiguous, this book sometimes refers to a program as being in a particular state. This means that one of the program's conversations (the one being discussed) is in a particular state.

Starter-Set Flows

This section provides examples of programs using the CPI-Communications starter-set calls:

- "Example 1: Data Flow in One Direction" on page 23 demonstrates a flow of data in only one direction (only the initiating program sends data).
- "Example 2: Data Flow in Both Directions" on page 26 describes a bidirectional flow of data (the initiating program sends data, and then allows the partner program to send data).

Example 1: Data Flow in One Direction

Figure 3 on page 25 shows an example of a conversation where the flow of data is in only one direction.

The steps shown in Figure 3 are described below:

Step	Description
1	To communicate with its partner program, Program A must first establish a conversation. Program A uses the Initialize_Conversation call to tell CPI Communications that it wants to: • Initialize a conversation • Identify the conversation partner (using *sym_dest_name*) • Ask CPI Communications to establish the identifier that the program will use when referring to the conversation (the *conversation_ID*). Upon successful completion of the Initialize_Conversation call, CPI Communications provides the *conversation_ID* and returns it to Program A. The program must store the *conversation_ID* and use it on all subsequent calls intended for that conversation.
2	No errors were found on the Initialize_Conversation call, and the *return_code* is set to CM_OK. Two major tasks are now accomplished: • CPI Communications has established a set of conversation characteristics for the conversation, based on the *sym_dest_name*, and uniquely associated them with the *conversation_ID*. • The default values for the conversation characteristics, as listed in "Initialize_Conversation (CMINIT)" on page 68, have been assigned to the characteristics. (For example, the conversation now has *conversation_type* set to CM_MAPPED_CONVERSATION.)
3	Program A asks that a conversation be started with an Allocate call (see "Allocate (CMALLC)" on page 49) using the *conversation_ID* previously assigned by the Initialize_Conversation call.
4	If a session between the LUs is not already available, one is activated. Program A and Program C can now have a conversation.
5	A *return_code* of CM_OK indicates that the Allocate call was successful and the LU has allocated the necessary resources to the program for its conversation. Program A's conversation is now in **Send** state and Program A can begin to send data. **Note:** In this example, the error conditions that can arise, such as no sessions available, are not discussed. See "Allocate (CMALLC)" on page 49 for more information about the error conditions that can result.

Step	Description
6 and 7	Program A sends data with the Send_Data call (described in "Send_Data (CMSEND)" on page 83) and receives a *return_code* of CM_OK. Until now, the conversation may not have been established because the conversation start-up request may not be sent until the first flow of data. In fact, any number of Send_Data calls can be issued before CPI Communications actually has a full buffer, which causes it to send the start-up request and data. Step 6 shows a case where the amount of data sent by the first Send_Data is greater than the size of the local LU's send buffer (a system dependent property), which is one of the conditions that triggers the sending of data. The request for a conversation is sent at this time. **Note:** Some products may choose to transmit the conversation start-up request as part of the Allocate processing. See the specific product documentation for details. For a complete discussion of transmission conditions and how to ensure the immediate establishment of a conversation and transmission of data, see "Data Buffering and Transmission" on page 29.
8 and 9	Once the conversation is established, the remote program's system takes care of starting Program C. The conversation on Program C's side is in **Reset** state and Program C issues a call to Accept_Conversation, which places the conversation into **Receive** state. The Accept_Conversation call is similar to the Initialize_Conversation call in that it equates a *conversation_ID* with a set of conversation characteristics (see "Accept_Conversation (CMACCP)" on page 47 for details). Program C, like Program A in Step 2, receives a unique *conversation_ID* that it will use in all future CPI-Communications calls for that particular conversation. As discussed in "Conversation Characteristics" on page 14, some of Program C's defaults are based on information contained in the conversation start-up request.
10 and 11	Once in **Receive** state, Program C begins whatever processing role it and Program A have agreed upon. In this case, Program C accepts data with a Receive call (as described in "Receive (CMRCV)" on page 74). Program A could continue to make Send_Data calls (and Program C continue to make Receive calls), but, for the purposes of this example, assume that Program A only wanted to send the data in its initial Send_Data.
12	Program A issues a Deallocate call (see "Deallocate (CMDEAL)" on page 56) to send any data buffered in the local LU and release the conversation. Program C issues a final Receive, shown here in the same step as the Deallocate, to check that it has all the received data.
13 and 14	The *return_code* of CM_DEALLOCATED_NORMAL tells Program C that the conversation is deallocated. Both Program C and Program A finish normally. **Note:** Only one program should issue Deallocate; in this case it was Program A. If Program C had issued Deallocate after receiving CM_DEALLOCATED_NORMAL, an error would have resulted.

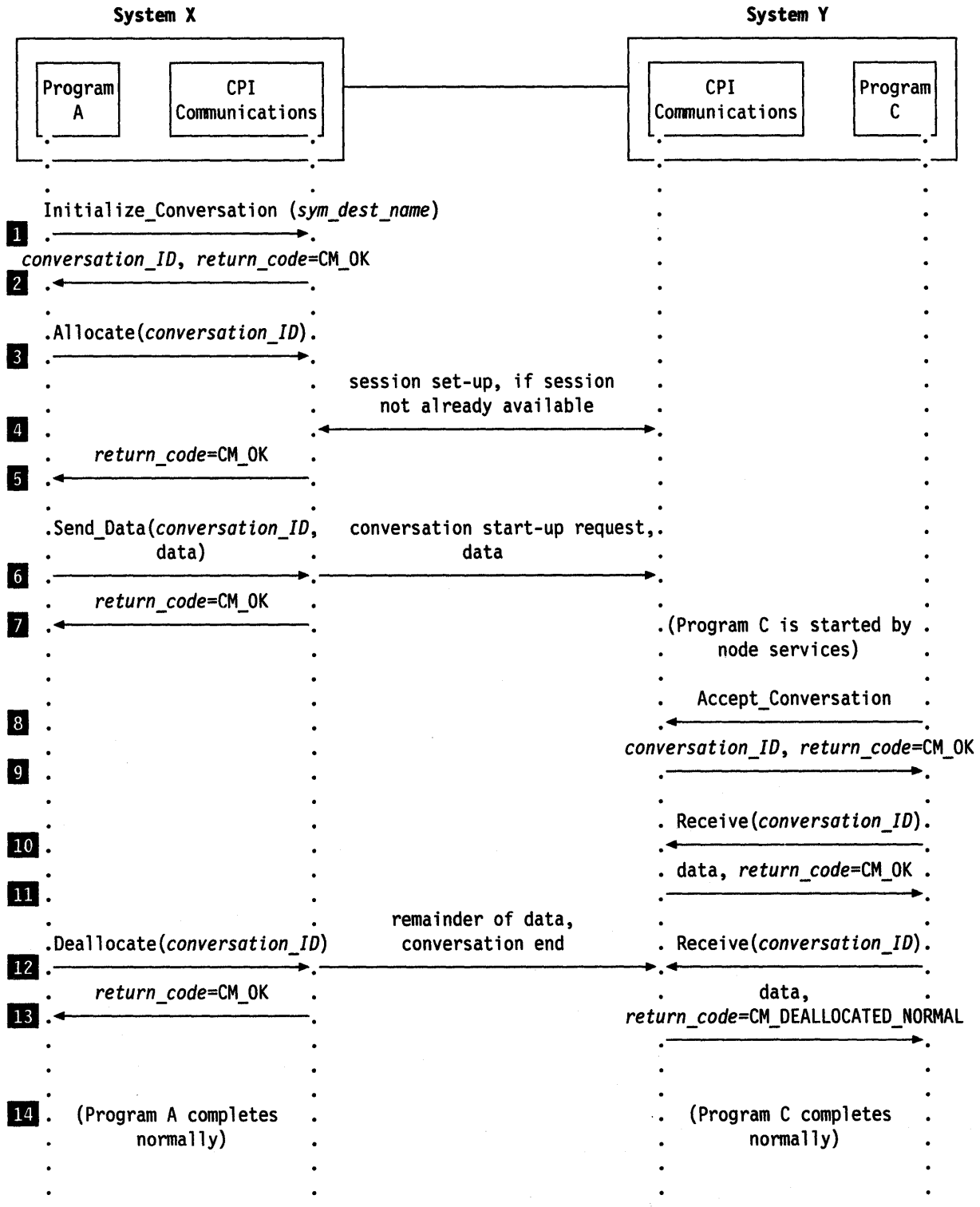


Figure 3. Data Flow in One Direction

Example 2: Data Flow in Both Directions

Figure 4 shows a conversation in which the flow of data is in both directions. It describes how two programs using starter-set calls can initiate a change of control over who is sending the data.

The steps shown in Figure 4 are described below:

Step	Description
1 through 4	Program A is sending data and Program C is receiving data. **Note:** The conversation in this example is already established with the default characteristics. Program A's end of the conversation is in **Send** state, and Program C's is in **Receive** state.
5	After sending some amount of data (an indeterminate number of Send_Data calls), Program A issues the Receive call while in **Send** state. As described in "Receive (CMRCV)" on page 74, this call causes the remaining data buffered at System X to be sent and permission to send to be given to Program C. Program A is placed in **Receive** state and waits until a response from Program C is received. **Note:** See "Example 3: The Sending Program Changes the Data Flow Direction" on page 30 for alternate methods that allow Program A to continue processing. Program C issues a Receive call in the same way it issued the two prior Receive calls.
6	Program C receives not only the last of the data from Program A, but also a *status_received* parameter set to CM_SEND_RECEIVED. The value of CM_SEND_RECEIVED notifies Program C that the conversation is now in **Send** state.
7	As a result of the *status_received* value, Program C issues a Send_Data call. The data from this call, on arrival at System X, is returned to Program A as a response to the Receive it issued in Step 5. At this point, the flow of data has been completely reversed and the two programs can continue whatever processing their logic dictates. To give control of the conversation back to Program A, Program C would simply follow the same procedure that Program A executed in Step 5.
8 through 10	Program A and Program C continue processing. Program C sends data and Program A receives the data.

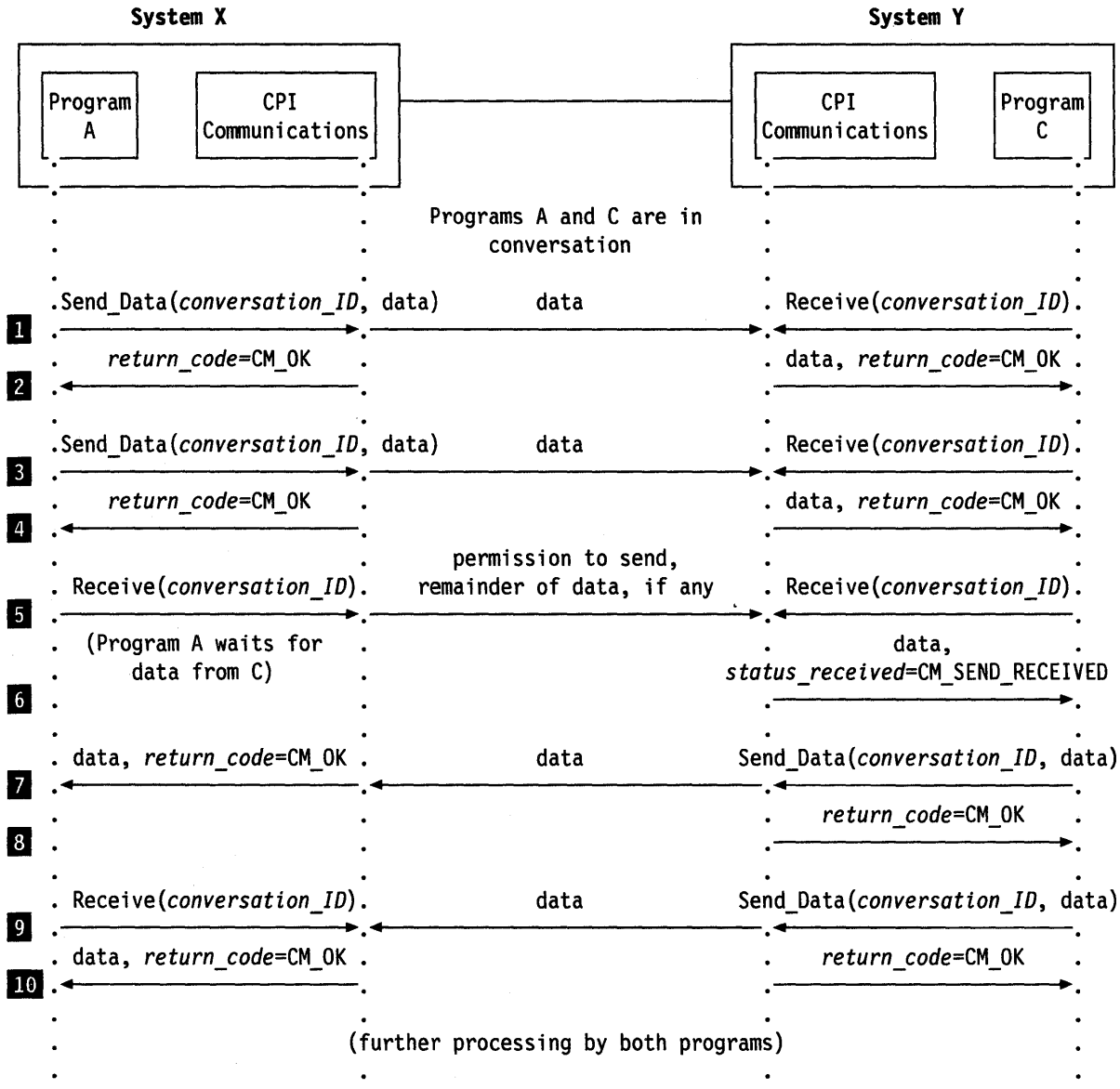


Figure 4. Data Flow in Both Directions

Advanced-Function Flows

This section provides examples of programs using the advanced-function calls:

- "Example 3: The Sending Program Changes the Data Flow Direction" shows how to use the Prepare_To_Receive call to change direction of data flow.
- "Example 4: Validation and Confirmation of Data Reception" shows how to use the Confirm and Confirmed calls to validate data reception. The Flush call is also shown.
- "Example 5: The Receiving Program Changes the Data Flow Direction" shows how to use the Request_To_Send call to request a change in the direction of data flow.
- "Example 6: Reporting Errors" shows how to use the Send_Error call to report errors in the data flow.
- "Example 7: Error Direction and Send-Pending State" shows how to use the **Send-Pending** state and the *error_direction* characteristic to resolve an ambiguous error condition that can occur when a program receives both a change-of-direction indication and data on a Receive call.

This section begins with a discussion of how a program can exercise control over when the LU actually transmits the data.

Data Buffering and Transmission

If a program uses the initial set of conversation characteristics, data is not automatically sent to the remote program after a Send_Data has been issued, except when the send buffer at the local LU overflows. As shown in the starter-set flows, the start-up of the conversation and subsequent data flow can occur any time after the program call to Allocate. This is because the LU stores the data in internal buffers and groups transmissions together for efficiency.

A program can exercise explicit control over the transmission of data by using one of the following calls to cause the buffered data's immediate transmission:

- Confirm
- Deallocate
- Flush
- Prepare_To_Receive
- Receive (in **Send** state) with *receive_type* set to CM_RECEIVE_AND_WAIT (*receive_type*'s default setting)
- Send_Error.

The use of Receive in **Send** state or Deallocate has already been shown in the starter-set flows. The other calls are discussed in the following examples.

Example 3: The Sending Program Changes the Data Flow Direction

Figure 5 is a variation on the function provided by the flow shown in "Example 2: Data Flow in Both Directions" on page 26. When the data flow direction changes, Program A can continue processing instead of waiting for data to arrive.

The steps shown in Figure 5 are described below:

Step	Description
1 through 6	The program begins the same as "Example 1: Data Flow in One Direction" on page 23. Program A establishes the conversation and makes the initial transmission of data.
7 through 10	Program A makes use of an advanced-function call, Prepare_To_Receive, (described in "Prepare_To_Receive (CMPTR)" on page 71), which sends an indication to Program C that Program A is ready to receive data. This call also flushes the data buffer and places Program A into **Receive** state. It does not, as did the Receive call when used with the initial conversation characteristics in effect, force Program A to pause and wait for data from Program C to arrive. Program A continues processing while data is sent to Program C.
11 through 13	Program C, started by System Y's reception of the conversation start-up request and buffered data, makes the Accept_Conversation and Receive calls. Program A finishes its processing and issues its own Receive call. It will now wait until data is received (Step 15).
14 through 16	The *status_received* on the Receive call made by Program C, which is set to CM_SEND_RECEIVED, tells Program C that the conversation is in **Send** state. Program C can now issue the Send_Data call. Program A receives the data. **Note:** There is a way for Program A to check periodically to see if the data has arrived, without waiting (not shown in the figure). After issuing the Prepare_To_Receive call, Program A can use the Set_Receive_Type call to set the *receive_type* conversation characteristic equal to CM_RECEIVE_IMMEDIATE. This call changes the nature of all subsequent Receives issued by Program A (until a further call to Set_Receive_Type is made). If a Receive is issued with the *receive_type* set to CM_RECEIVE_IMMEDIATE, the program retains control of processing without waiting. It receives data back if data is present, and a *return_code* of CM_UNSUCCESSFUL if no data has arrived. For further discussion of this alternate flow, see "Set_Receive_Type (CMSRT)" on page 110 and "Receive (CMRCV)" on page 74.

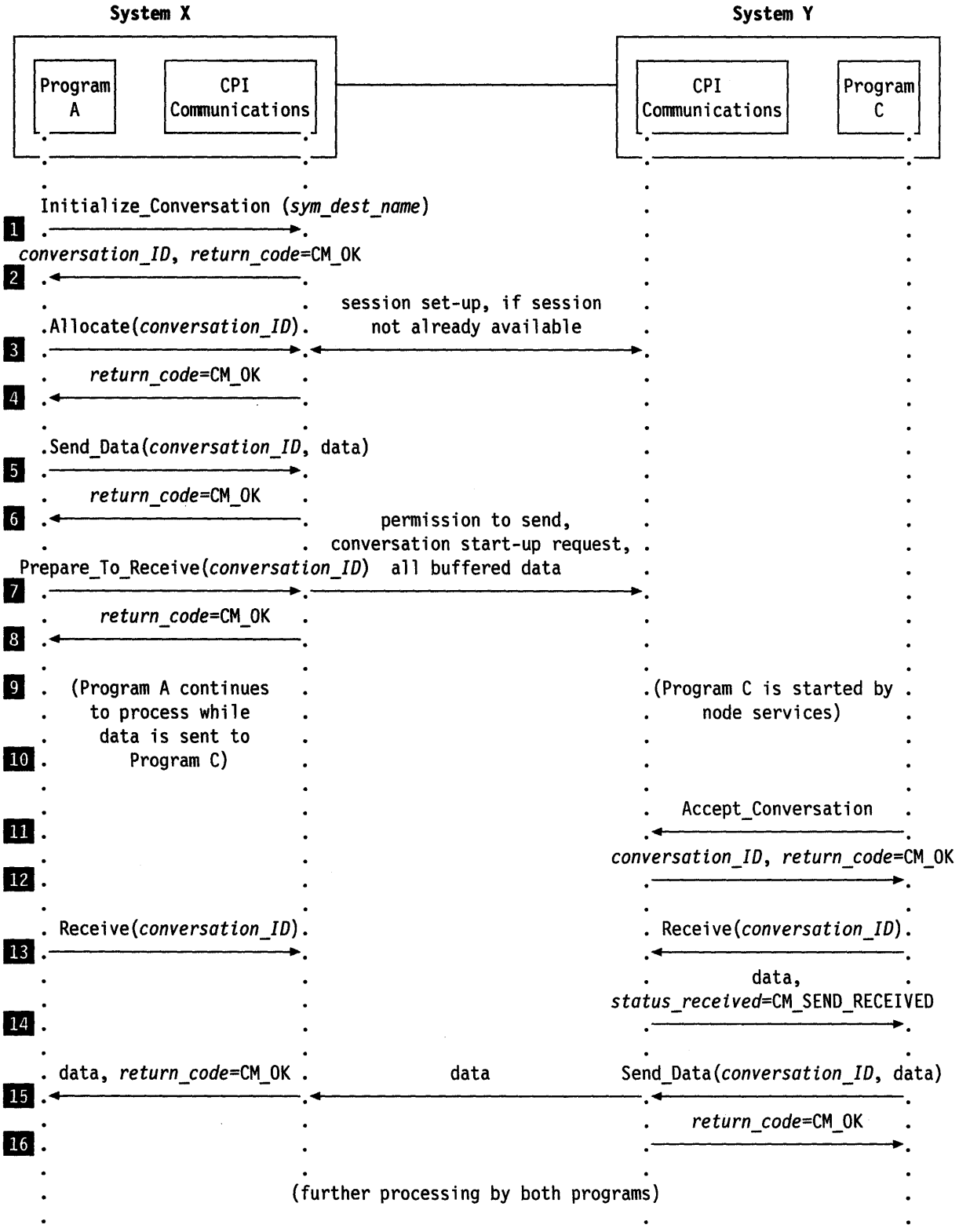


Figure 5. The Sending Program Changes the Data Flow Direction

Example 4: Validation and Confirmation of Data Reception

Figure 6 on page 33 shows how a program can use the Confirm and Confirmed calls to verify receipt of its sent data. The Flush call is also shown.

The steps shown in Figure 6 are described below:

Step	Description
1 and 2	As before, Program A issues the Initialize_Conversation call to initialize the conversation.
3 and 4	Program A issues a new call, Set_Sync_Level(CM_CONFIRM) to set the *sync_level* characteristic. **Note:** Program A must set the *sync_level* characteristic before issuing the Allocate call (Step 5) for the value to take effect. Changing the *sync_level* after the conversation is allocated results in an error condition. See "Set_Sync_Level (CMSSL)" on page 116 for a detailed discussion of the *sync_level* characteristic and the meaning of CM_CONFIRM.
5 and 6	Program A issues the Allocate call to start the conversation.
7 and 8	Program A uses the Flush call (see "Flush (CMFLUS)" on page 66) to make sure that the conversation is immediately established. If data is present, the local LU buffer is emptied and the contents sent to the remote LU. Since no data is present, only the LU 6.2 conversation start-up request is sent to establish the conversation. At System Y, the conversation start-up request is received. Program C is started and begins processing.
9 and 10	Program A issues a Send_Data call. Program C issues an Accept_Conversation call.
11	Program A issues a Confirm call to make sure that Program C has received the data, and is forced to wait for a reply. As a result of the Confirm call, the local LU flushes its send buffer and the data is immediately transmitted to the remote LU.
12 and 13	Program C issues a Receive call and receives the data with *status_received* set to CM_CONFIRM_RECEIVED.
14 and 15	Because *status_received* is set to CM_CONFIRM_RECEIVED, indicating a confirmation request, the conversation has been placed into **Confirm** state. Program C must now issue a Confirmed call. After Program C makes the Confirmed call (see "Confirmed (CMCFMD)" on page 54), the conversation returns to **Receive** state. Meanwhile, at System X, the confirmation reply arrives and the CM_OK *return_code* is sent back to Program A.
16	Program A continues with further processing. **Note:** Unlike the previous examples in which a program could bypass waiting, this example demonstrates that use of the Confirm call forces the program to wait for a reply.

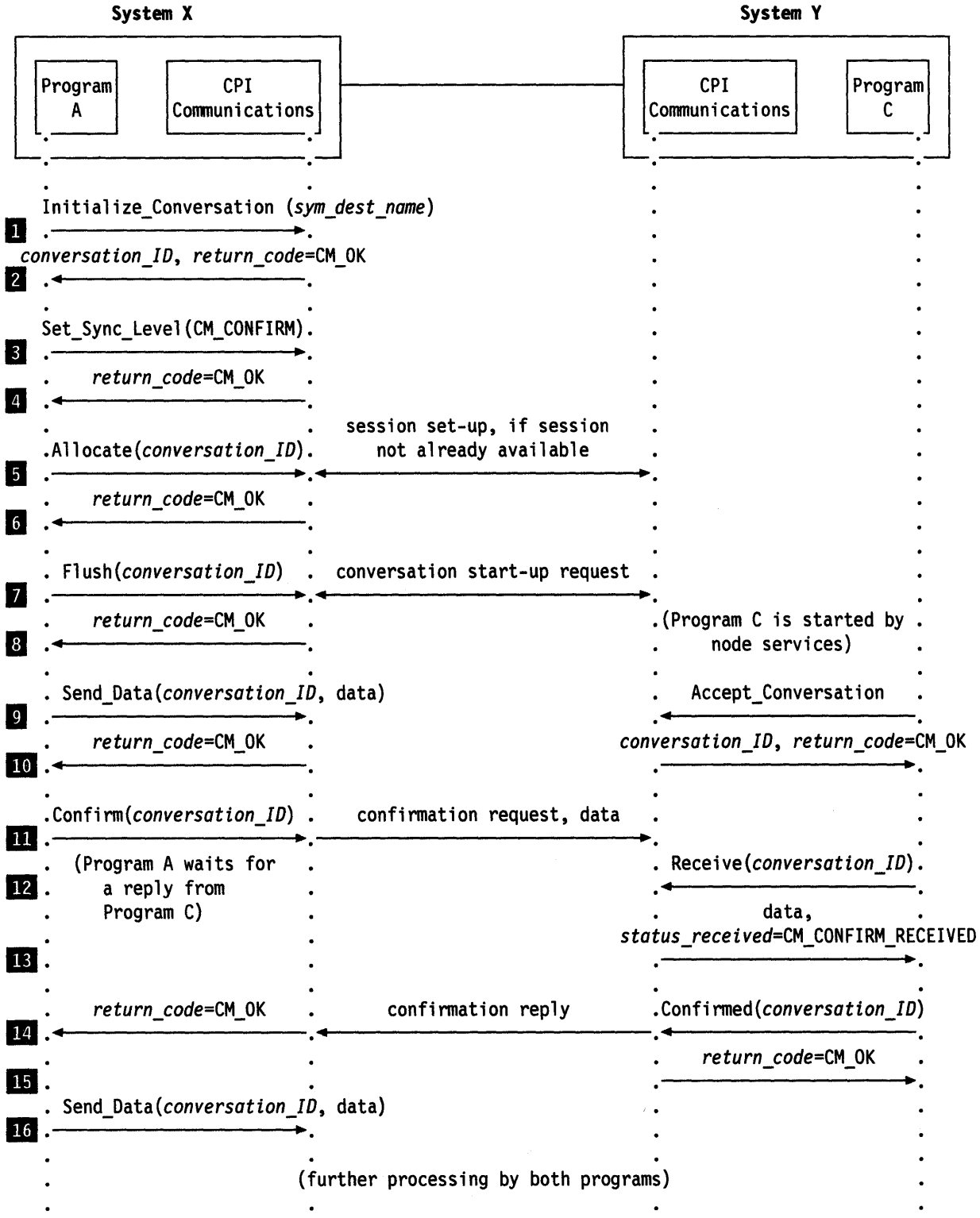


Figure 6. Validation and Confirmation of Data Reception

Example 5: The Receiving Program Changes the Data Flow Direction

Figure 7 shows how a program on the receiving side of a conversation can request a change in the direction of data flow with the Request_To_Send call. (See "Request_To_Send (CMRTS)" on page 81 for more information.) In this example, Programs A and C have already established a conversation using the default conversation characteristics.

The steps shown in Figure 7 are described below:

Step	Description
1 and 2	Program A is sending data and Program C is receiving the data.
3 and 4	Program C issues a Request_To_Send call in order to begin sending data. Program A will be notified of this request on the return value of the next call (Send_Data in this case, Step 6) issued by Program A.
5 and 6	Program A issues a Send_Data request, and the call returns with *request_to_send_received* set equal to CM_REQUEST_TO_SEND_RECEIVED.
7 and 8	In reply to the Request_To_Send, Program A issues a Prepare_To_Receive call, which allows Program A to continue its own processing and passes permission to send to Program C. The call also forces the buffers at System X to be flushed. It leaves the conversation in **Receive** state for Program A. **Note:** Program A does not have to reply to the Request_To_Send call immediately (as it does in this example). See "Example 3: The Sending Program Changes the Data Flow Direction" on page 30 for other possible responses. Program C continues with normal processing by issuing a Receive call and receives Program A's acceptance of the Request_To_Send on the *status_received* parameter, which is set to CM_SEND_RECEIVED. The conversation is now in **Send** state for Program C.
9 and 10	Program C can now transmit data. Because Program C has only one instance of data to transmit, it first changes the *send_type* conversation characteristic by issuing Set_Send_Type. Setting *send_type* to a value of CM_SEND_AND_PREP_TO_RECEIVE means that Program C will return to **Receive** state after issuing a Send_Data call. It also forces a flushing of the LU's data buffer.
11	Program C issues the Send_Data call and is placed in **Receive** state. The data and permission-to-send indication are transmitted from System Y to System X. Program A, meanwhile, has finished its own processing and issued a (perfectly-timed, in this diagram) Receive call.
12	Program A receives the data requested and, because of the value of the *status_received* parameter (which is set to CM_SEND_RECEIVED), knows that the conversation has been returned to **Send** state.
13 and 14	The original processing flow continues: Program A issues a Send_Data call and Program C issues a Receive call.

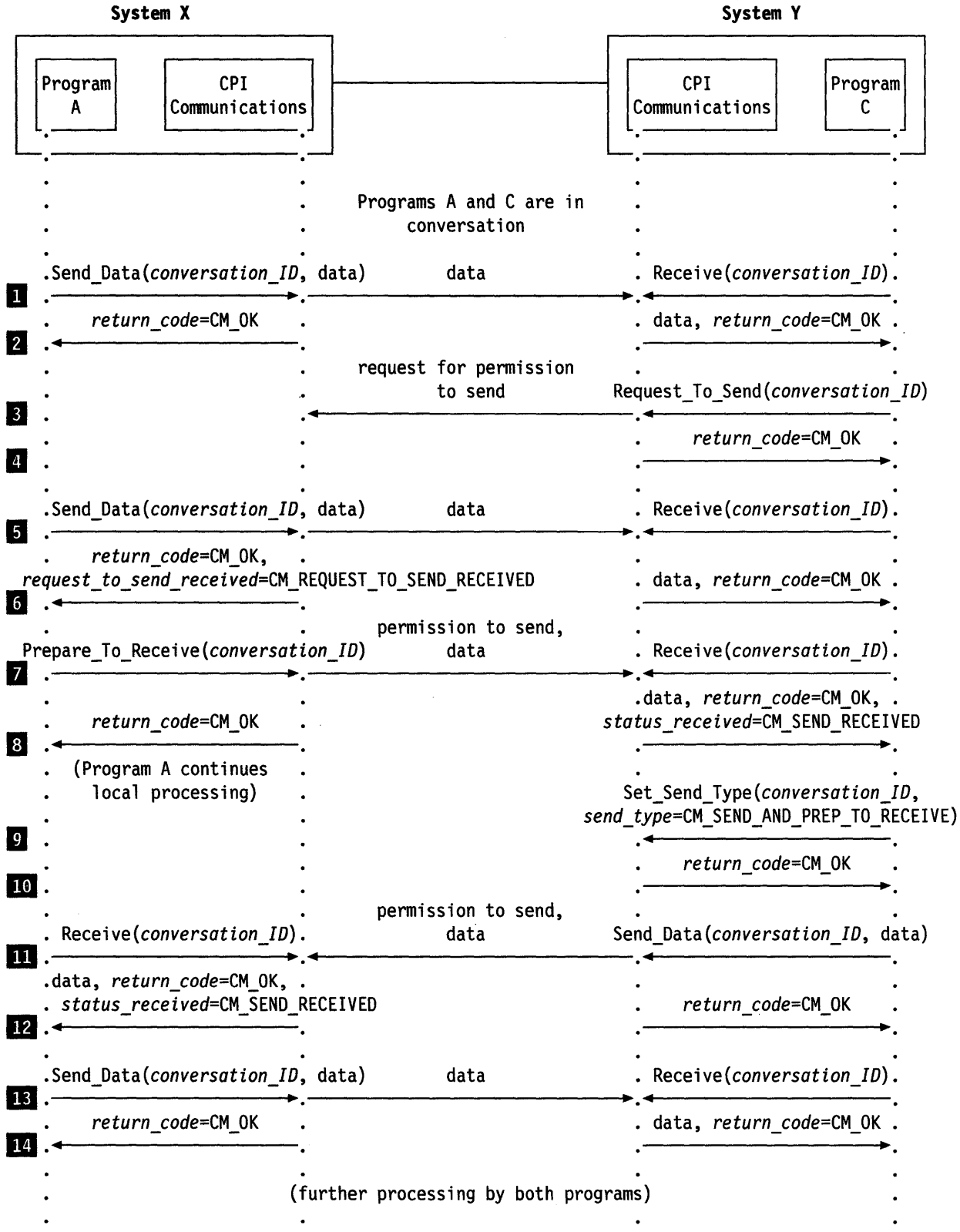


Figure 7. The Receiving Program Changes the Data Flow Direction

Example 6: Reporting Errors

All the previous examples assumed that no errors were found in the data, and that the receiving program was able to continue receiving data. However, in some cases the local program may detect an error in the data or may find that it is unable to receive more data (for example, its buffers are full) and cannot wait for the remote program to honor a request-to-send request. Figure 8 shows how to use the Send_Error call in these situations.

Note: This example describes the simplest type of error-reporting, an error found while receiving data. "Example 7: Error Direction and Send-Pending State" on page 38 describes a more complicated use of Send_Error.

The steps shown in Figure 8 are described below:

Step	Description
1 and 2	Program A is sending data and Program C is receiving data. The initial characteristic values set by Initialize_Conversation and Accept_Conversation are in effect.
3 and 4	Program C encounters an error on the received data and issues the Send_Error call. The local LU sends control information to the LU at System X indicating that the Send_Error has been issued and purges all data contained in its buffer.
5 and 6	Meanwhile, Program A has sent more data. This data is lost because the LU at System X knows that a Send_Error has been issued at System Y (the control information sent in Step 3). After the LU at System X sends control information to System Y, a *return_code* of CM_OK is returned to Program C and the conversation is left in **Send** state. Program A learns of the error (and possibly lost data) when it receives back the *return_code*, which is set to CM_PROGRAM_ERROR_PURGING. Program A's end of the conversation is also, in a parallel action to the now-new **Send** state of the conversation for Program C, placed into **Receive** state.
7 and 8	Program C issues a Send_Data call, and Program A receives the data using the Receive call. Programs A and C continue processing normally.

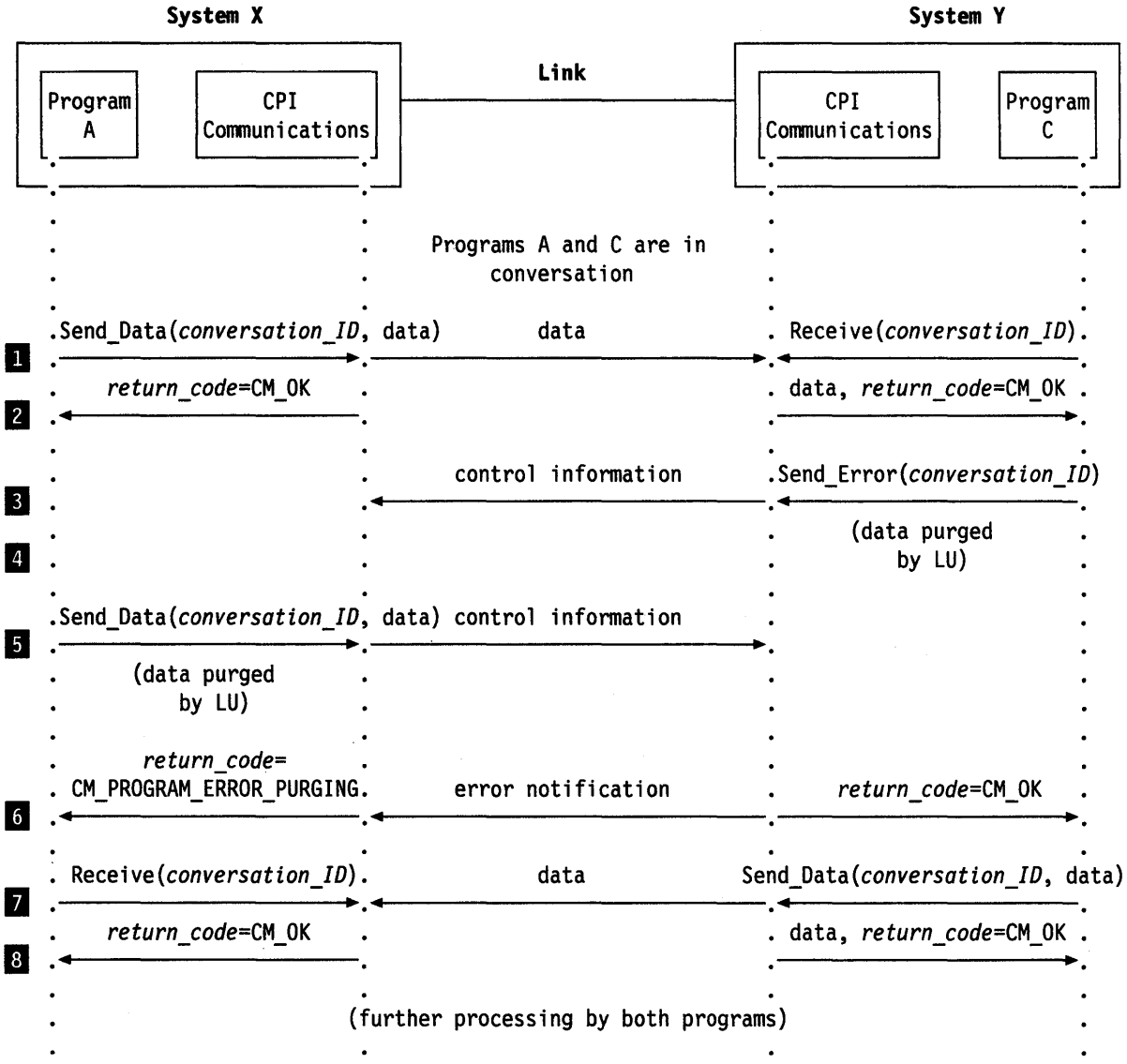


Figure 8. Reporting Errors

Example 7: Error Direction and Send-Pending State

Figure 9 on page 39 shows how to use the **Send-Pending** state and the *error_direction* characteristic to resolve an ambiguous error condition that can occur when a program receives both a change of direction indication and data on a Receive call.

The steps shown in Figure 9 are described below:

Step	Description
1 and 2	The conversation has already been established using the default conversation characteristics. Program A is sending data in **Send** state and Program C is receiving data in **Receive** state.
3	Program A issues the Receive call to begin receiving data and enters **Receive** state.
4 and 5	Program C issues a Receive and is notified of the change in the conversation's state by the *status_received* parameter, which is set to CM_SEND_RECEIVED. The reception of both data and CM_SEND_RECEIVED on the same Receive call places Program C into **Send-Pending** state. Two possible error conditions can now occur: • Program C, while processing the data just received, discovers something wrong with the data (as was discussed in "Example 6: Reporting Errors"). This is an error in the "receive" direction of the data. • Program C finishes processing the data and begins its send processing. However, it discovers that it cannot send a reply. For example, the received data might contain a query for a particular data base. Program C successfully processes the query but, on attempting to access that data base, finds that the data base is not available. This is an error in the "send" direction of the data. The *error_direction* characteristic is used to indicate which of these two conditions has occurred. A program sets *error_direction* to CM_RECEIVE_ERROR for the first case and sets *error_direction* to CM_SEND_ERROR for the second.
6 and 7	In this example, Program C encounters a send error and issues Set_Error_Direction to set the *error_direction* characteristic to CM_SEND_ERROR. **Note:** The *error_direction* characteristic was not set in the previous example because the initial value is CM_RECEIVE_ERROR, which accurately describes the error encountered in that example.
8	Program C issues Send_Error. Because CPI Communications knows the program is in **Send-Pending** state, it checks the *error_direction* characteristic and notifies the CPI-Communications component at System X which type of error has occurred. Program A receives the error information in the *return_code*. The *return_code* is set to CM_PROGRAM_NO_TRUNC because Program C set *error_direction* to CM_SEND_ERROR. If the error_direction had been set to CM_RECEIVE_ERROR, Program A would have received a *return_code* of CM_PROGRAM_ERROR_PURGING (as in the previous example).
9 through 11	Program C notifies Program A of the exact nature of the problem and both programs continue processing.

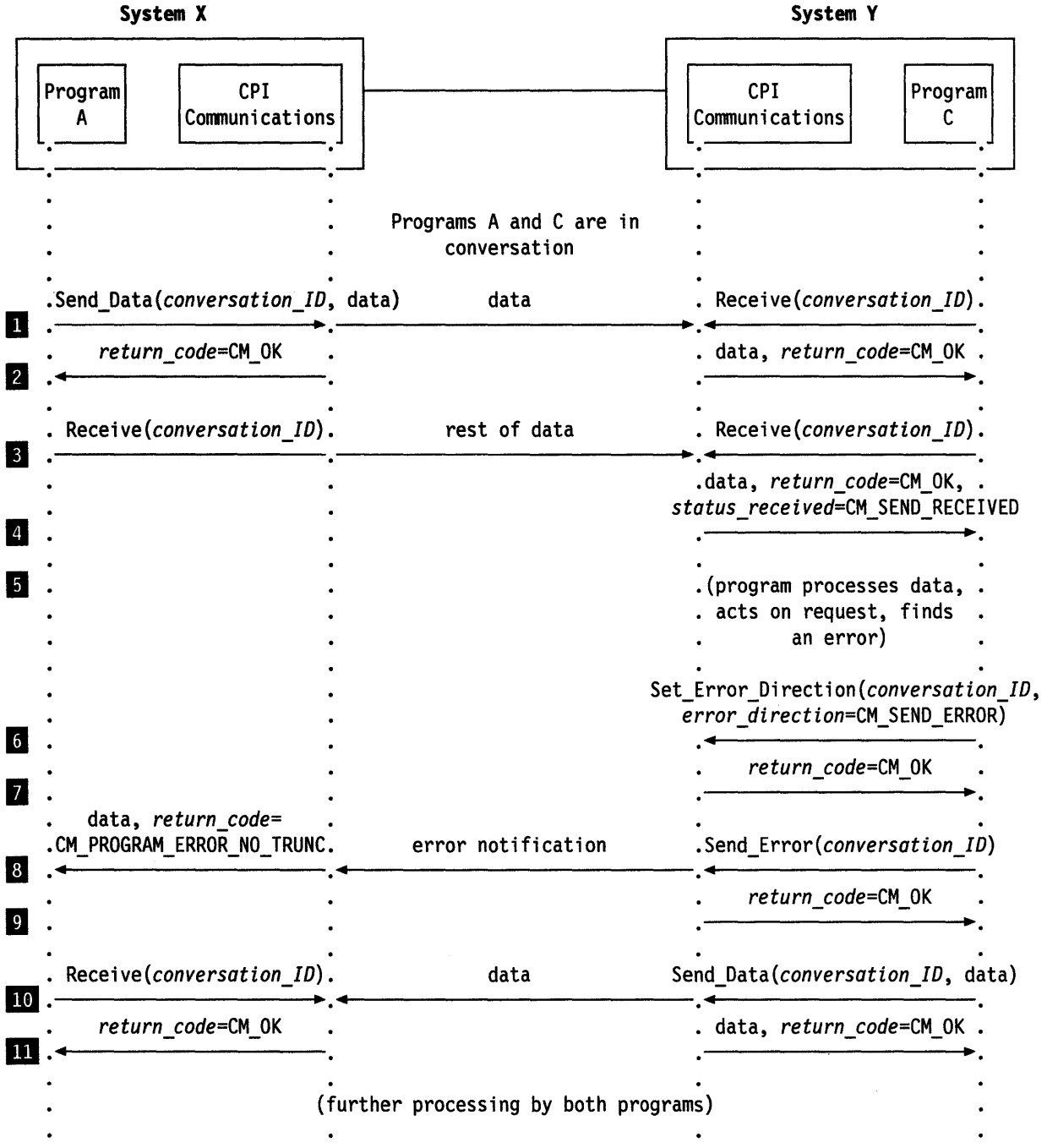


Figure 9. Error Direction and Send-Pending State

Chapter 4. Reference Section

This chapter describes the CPI-Communications calls. For each call, this chapter provides the function of the call and any optional setup calls, which can be issued before the call being described. In addition, the following information is provided if it applies:

- **Format**

  The format used to program the calls.

  Note: The actual syntax used to program the calls in this chapter depends on the programming language used. See "Call Syntax" on page 42 for specifics.

- **Parameters**

  The parameters that are required for the calls.

- **State Changes**

  The changes in the conversation state that can result from this call. See "Program Flow — States and Transitions" on page 17 for more information on program states.

  Note: When the meaning is unambiguous, this book sometimes refers to a program as being in a particular state. This means that one of the program's conversations (the one being discussed) is in a particular state.

- **Usage Notes**

  Additional information that applies to the call.

- **Related Information**

  Where to find additional information related to the call.

Call Syntax

CPI-Communications calls can be made from application programs written in a number of high-level programming languages:

- C
- COBOL
- FORTRAN
- REXX (SAA Procedures Language)
- CSP (SAA Application Generator)

Specific syntax and library information for VM can be found in Appendix E, "CMS VM/SP—Extension Information"

This book uses a general call format to show the name of the CPI-Communications call and parameters used. An example of that format is provided below:

```
CALL CMPROG (parm0,
             parm1,
             parm2,
               .
               .
             parmN)
```

where CMPROG is the name of the call, and parm0, parm1, parm2, and parmN represent the parameter list described in the individual call descriptions.

This format would be translated into, for each of the supported languages, the following syntaxes:

C

```
CMPROG (parm0,parm1,parm2,...parmN)
```

COBOL

```
CALL "CMPROG" USING parm0,parm1,parm2,...parmN
```

CSP

```
CALL CMPROG parm0,parm1,parm2,...parmN
```

FORTRAN

```
CALL CMPROG (parm0,parm1,parm2,...parmN)
```

REXX

```
ADDRESS CPICOMM 'CMPROG parm0 parm1 parm2 ... parmN'
```

"Programming Language Considerations" on page 177 in Appendix E, "CMS VM/SP—Extension Information" provides further information on the different programming languages and how they can be used in the CMS environment.

How to Use the Call References

Here is an example of how the information in this chapter can be used in connection with the material in the rest of the book. The example describes how to use the Set_Return_Control call to set the conversation characteristic of *return_control* to a value of CM_IMMEDIATE.

- "Set_Return_Control (CMSRC)" on page 112 contains the semantics of the variables used for the call. It explains that the real name of the program call for Set_Return_Control is CMSRC and that CMSRC has a parameter list of *conversation_ID*, *return_control*, and *return_code*.
- "Call Syntax" on page 42 shows the syntax for the programming language being used.
- Appendix A, "Variables and Characteristics" provides a complete description of all variables used in the book and shows that the *return_control* variable that goes into the call as a parameter is a 32-bit integer. This information is provided in Table 6 on page 129.
- Table 4 on page 124 in Appendix A, "Variables and Characteristics" shows that the value of CM_IMMEDIATE that is placed into the *return_control* parameter on the call to CMSRC is defined as an integer value of 1.
- Finally, the meaning of the *return_code* value CM_OK that is returned to the program on the call is provided in Appendix B, "Return Codes." It means that the call completed successfully.

Locations of Key Topics

A list of program calls by their call names — providing the call pseudonym, a brief description, and the call's location in this chapter — is given on page 44. A list of key-topic discussions and where they occur is provided below:

- "Naming Conventions — Calls and Characteristics, Variables and Values" on page 19 describes the naming conventions used throughout the book.
- "Data Buffering and Transmission" on page 29 provides a discussion of program control over data transmission.
- "Usage Notes" of "Request_To_Send (CMRTS)" on page 81 discusses how a program enters **Receive** state.
- "Usage Notes" of "Send_Data (CMSEND)" on page 85 describes the use of logical records and LL fields on basic conversations.

Call	Pseudonym	Description	Page
CMACCP	Accept_Conversation	Used by a program to accept an incoming conversation.	47
CMALLC	Allocate	Used by a program to establish a conversation.	49
CMCFM	Confirm	Used by a program to send a confirmation request to its partner.	52
CMCFMD	Confirmed	Used by a program to send a confirmation reply to its partner.	54
CMDEAL	Deallocate	Used by a program to end a conversation.	56
CMECT	Extract_Conversation_Type	Used by a program to view the current *conversation_type* conversation characteristic.	59
CMEMN	Extract_Mode_Name	Used by a program to view the current *mode_name* conversation characteristic.	60
CMEPLN	Extract_Partner_LU_Name	Used by a program to view the current *partner_LU_name* conversation characteristic.	62
CMESL	Extract_Sync_Level	Used by a program to view the current *sync_level* conversation characteristic.	64
CMFLUS	Flush	Used by a program to flush the LU's send buffer.	66
CMINIT	Initialize_Conversation	Used by a program to initialize the conversation characteristics.	68
CMPTR	Prepare_To_Receive	Used by a program to change a conversation from **Send** to **Receive** state in preparation to receive data.	71
CMRCV	Receive	Used by a program to receive data.	74
CMRTS	Request_To_Send	Used by a program to notify its partner that it would like to send data.	81
CMSCT	Set _Conversation_Type	Used by a program to set the *conversation_type* conversation characteristic.	93

Call	Pseudonym	Description	Page
CMSDT	Set_Deallocate_Type	Used by a program to set the *deallocate_type* conversation characteristic.	95
CMSED	Set_Error_Direction	Used by a program to set the *error_direction* conversation characteristic.	98
CMSEND	Send_Data	Used by a program to send data.	83
CMSERR	Send_Error	Used by a program to notify its partner of an error that occurred during the conversation.	88
CMSF	Set_Fill	Used by a program to set the *fill* conversation characteristic.	100
CMSLD	Set_Log_Data	Used by a program to set the *log_data* conversation characteristic.	102
CMSMN	Set_Mode_Name	Used by a program to set the *mode_name* conversation characteristic.	104
CMSPLN	Set_Partner_LU_Name	Used by a program to set the *partner_LU_name* conversation characteristic.	106
CMSPTR	Set_Prepare_To_Receive_Type	Used by a program to set the *prepare_to_receive_type* conversation characteristic.	108
CMSRC	Set_Return_Control	Used by a program to set the *return_control* conversation characteristic.	112
CMSRT	Set_Receive_Type	Used by a program to set the *receive_type* conversation characteristic.	110
CMSSL	Set_Sync_Level	Used by a program to set the *sync_level* conversation characteristic.	116
CMSST	Set_Send_Type	Used by a program to set the *send_type* conversation characteristic.	114

Call	Pseudonym	Description	Page
CMSTPN	Set_TP_Name	Used by a program to set the *TP_Name* conversation characteristic.	118
CMTRTS	Test_Request_To_Send_Received	Used by a program to determine whether or not the remote program is requesting to send data.	120

Accept_Conversation (CMACCP)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

The Accept_Conversation call accepts an incoming conversation. Like Initialize_Conversation (CMINIT), this call initializes values for various conversation characteristics. The difference between the two calls is that the program that will later allocate the conversation issues the Initialize_Conversation call, and the partner program that will accept the conversation after it is allocated issues the Accept_Conversation call.

When the Accept_Conversation call completes successfully, the following conversation characteristics are initialized:

Conversation Characteristic	Initialized Value
conversation_type	Derived from the received conversation start-up request
deallocate_type	CM_DEALLOCATE_SYNC_LEVEL
error_direction	CM_RECEIVE_ERROR
fill	CM_FILL_LL
log_data	Null value
log_data_length	0
mode_name	Mode name of session over which the conversation start-up request was received
mode_name_length	Length of mode name
partner_LU_name	Partner LU name of session over which the conversation start-up request was received
partner_LU_name_length	Length of partner LU name
prepare_to_receive_type	CM_PREP_TO_RECEIVE_SYNC_LEVEL
receive_type	CM_RECEIVE_AND_WAIT
return_control	Null value **Note:** This characteristic applies only to an Allocate call.
send_type	CM_BUFFER_DATA
sync_level	Derived from the received conversation start-up request
TP_name	Null value **Note:** This characteristic applies only to an Allocate call.
TP_name_length	0

Format

```
CALL CMACCP(conversation_ID,
            return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier assigned to the conversation. CPI Communications supplies and maintains the *conversation_ID*. When the *return_code* is set equal to CM_OK, the value returned in this parameter is used by the program on all subsequent calls issued for this conversation.

return_code
Specifies the result of the call execution, which is returned to the local program. The *return_code* variable can have one of the following values:

- CM_OK
- CM_PROGRAM_STATE_CHECK
  This value indicates that no incoming conversation exists.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

When *return_code* is set equal to CM_OK, the local program enters the **Receive** state.

Usage Notes

For each conversation, CPI Communications assigns a unique identifier (the *conversation_ID*) that the program uses in all future calls intended for that conversation. Therefore, the partner program must issue the Accept_Conversation call before any other calls can refer to the conversation.

Related Information

"Conversation Characteristics" on page 14 provides a comparison of conversation characteristics as set by Accept_Conversation and Initialize_Conversation.

"Example 1: Data Flow in One Direction" on page 23 shows an example program flow using the Accept_Conversation call.

"Initialize_Conversation (CMINIT)" on page 68 describes how the conversation characteristics are initialized for the program that allocates the conversation.

Allocate (CMALLC)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

A program uses the Allocate (CMALLC) call to establish a basic or mapped conversation (depending on the *conversation_type* characteristic) with its partner program. The partner program is specified in the *TP_name* characteristic.

Note: A *return_code* of CM_OK indicates that the conversation has been successfully allocated.

Optional set-up:

CALL CMSCT – Set_Conversation_Type
CALL CMSDT – Set_Deallocate_Type
CALL CMSMN – Set_Mode_Name
CALL CMSPLN – Set_Partner_LU_Name
CALL CMSRC – Set_Return_Control
CALL CMSRT – Set_Receive_Type
CALL CMSSL – Set_Sync_Level
CALL CMSST – Set_Send_Type
CALL CMSTPN – Set_TP_Name

Format

```
CALL CMALLC(conversation_ID,
            return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier of an initialized conversation.

return_code
Specifies the result of the call execution, which is returned to the local program. The *return_control* characteristic determines which return codes can be returned to the local program.

If *return_control* is set to CM_WHEN_SESSION_ALLOCATED, *return_code* can have one of the following values:

- CM_OK
- CM_ALLOCATE_FAILURE_NO_RETRY
- CM_ALLOCATE_FAILURE_RETRY
- CM_PARAMETER_ERROR
  This value indicates one of the following:
  – The *mode_name* characteristic (set from side information or by Set_Mode_Name) specifies a mode name that is not recognized by the LU as being valid.

  - The *mode_name* characteristic (set from side information or by Set_Mode_Name) specifies SNASVCMG, but the local program is not an SNA services program.
  - The *TP_name* characteristic (set from side information or by Set_TP_Name) specifies an SNA service transaction program name, but the local program does not have the appropriate privilege to allocate a conversation to an SNA service program.
  - The *TP_name* characteristic (set from side information or by Set_TP_Name) specifies an SNA service transaction program and *conversation_type* is set to CM_MAPPED_CONVERSATION.
  - The *partner_LU_name* characteristic (set from side information or by Set_Partner_LU_Name) specifies a partner LU name that is not recognized by the LU as being valid.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is not in **Initialize** state.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PRODUCT_SPECIFIC_ERROR

If *return_control* is set to CM_IMMEDIATE, *return_code* can have one of the following values:

- CM_OK
- CM_PARAMETER_ERROR
  This value indicates one of the following:
  - The *mode_name* characteristic (set from side information or by Set_Mode_Name) specifies a mode name that is not recognized by the LU.
  - The *mode_name* characteristic (set from side information or by Set_Mode_Name) specifies SNASVCMG, but the local program is not an SNA services program.
  - The *TP_name* characteristic (set from the side information or Set_TP_Name) specifies an SNA service transaction program name, but the local program does not have the appropriate privilege to allocate a conversation to an SNA service program.
  - The *TP_name* characteristic (set from the side information or Set_TP_Name) specifies an SNA service transaction program and the *conversation_type* is set to CM_MAPPED_CONVERSATION.
  - The *partner_LU_name* characteristic (set from side information or by Set_Partner_LU_Name) specifies a partner LU name that is not recognized by the LU as being valid.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is not in **Initialize** state.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PRODUCT_SPECIFIC_ERROR
- CM_UNSUCCESSFUL
  This value indicates that the session is not immediately available.

State Changes

When *return_code* is set equal to CM_OK, the local program enters **Send** state.

Usage Notes

1. An allocation error resulting from the local LU's failure to obtain a session for the conversation is reported on the Allocate call. An allocation error resulting from the remote LU's rejection of the allocation request is reported on a subsequent conversation call.
2. For CPI Communications to establish the conversation, CPI Communications must first establish a session between the local LU and the remote LU, if such a session does not already exist.
3. Depending on the circumstances, the local LU can send the conversation allocation request to the remote LU as soon as it allocates a session for the conversation. The local LU can also buffer the allocation request until it accumulates enough information for transmission (from one or more subsequent Send_Data calls), or until the local program issues a subsequent call other than Send_Data that explicitly causes the LU to flush its send buffer. The amount of information sufficient for transmission depends on the characteristics of the session allocated for the conversation and can vary from one session to another.
4. The local program can ensure that the remote program is connected as soon as possible by issuing Flush (CMFLUS) immediately after Allocate (CMALLC).
5. Security information for the LU and conversation is determined by the implementing product, not by CPI Communications.
6. After making a call to Accept_Conversation, the remote program is connected in **Receive** state.

Related Information

"Example 1: Data Flow in One Direction" on page 23 shows an example program flow using the Allocate call.

"SNA Service Transaction Programs" on page 149 provides a discussion of SNA service transaction programs.

"Data Buffering and Transmission" on page 29 provides a complete discussion of control methods for data transmission.

"Set_Return_Control (CMSRC)" on page 112 provides a discussion of the *return_control* characteristic.

"Set_Conversation_Type (CMSCT)" on page 93 provides a discussion of the *conversation_type* characteristic.

Confirm (CMCFM)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

The Confirm (CMCFM) call is used by a local program to send a confirmation request to the remote program and then wait for a reply. The remote program replies with a Confirmed (CMCFMD) call. The local and remote programs use the Confirm and Confirmed calls to synchronize their processing of data.

Note: The *sync_level* conversation characteristic for the *conversation_ID* specified must be set to CM_CONFIRM to use this call.

Format

```
CALL CMCFM(conversation_ID,
           request_to_send_received,
           return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier.

request_to_send_received
Specifies the variable in which is returned an indication of whether or not a request-to-send notification has been received. The *request_to_send_received* variable can have one of the following values:

- CM_REQ_TO_SEND_RECEIVED
  The local program received a request-to-send notification from the remote program. The remote program issued a Request_To_Send requesting that the local program enter **Receive** state, which places the remote program in **Send** state.
- CM_REQ_TO_SEND_NOT_RECEIVED
  A request-to-send notification has not been received.

Note: When *return_code* indicates CM_PROGRAM_PARAMETER_CHECK or CM_PROGRAM_STATE_CHECK, *request_to_send_received* does not contain a value.

return_code
Specifies the result of the call execution, which is returned to the local program. The *return_code* variable can have one of the following values:

- CM_OK (remote program replied Confirmed)
- CM_CONVERSATION_TYPE_MISMATCH
- CM_SECURITY_NOT_VALID
- CM_SYNC_LVL_NOT_SUPPORTED_PGM
- CM_TPN_NOT_RECOGNIZED
- CM_TP_NOT_AVAILABLE_NO_RETRY
- CM_TP_NOT_AVAILABLE_RETRY
- CM_DEALLOCATED_ABEND
- CM_PROGRAM_ERROR_PURGING
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY

- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The conversation is not in **Send** or **Send-Pending** state.
  - The conversation is basic and the program is in **Send** state, and the program started but did not finish sending a logical record.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *sync_level* conversation characteristic is set to CM_NONE.
  - The *conversation_ID* specifies an unassigned conversation identifier.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

When *return_code* is set to CM_OK:

- The program enters **Send** state if it issued the Confirm call in **Send-Pending** state.
- No state change occurs if the program issued the Confirm call in **Send** state.

Usage Notes

1. The program that issues Confirm must wait until a reply from the remote partner program (a reply made using the Confirmed call) is received.
2. The program can use this call for various application-level functions. For example:
   - The program can issue this call immediately following an Allocate call to determine if the conversation was allocated before sending any data.
   - The program can issue this call to determine if the remote program received the data sent. The remote program can respond by issuing a Confirmed call if it received and processed the data without error, or by issuing a Send_Error call if it encountered an error. The only other valid response from the remote program is the issuance of the Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND.
3. The send buffer of the local LU is flushed as a result of this call.

Related Information

"Confirmed (CMCFMD)" on page 54 provides information on the remote program's reply to the Confirm call.

"Request_To_Send (CMRTS)" on page 81 provides a complete discussion of the *request_to_send_received* parameter.

"Set_Sync_Level (CMSSL)" on page 116 explains how programs specify the level of synchronization processing.

"Example 4: Validation and Confirmation of Data Reception" on page 32 shows an example program using the Confirm call.

Confirmed (CMCFMD)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

A program uses the Confirmed (CMCFMD) call to send a confirmation reply to the remote program. The local and remote programs can use the Confirmed and Confirm calls to synchronize their processing.

Format

```
CALL CMCFMD(conversation_ID,
            return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier.

return_code
Specifies the result of the call execution, which is returned to the local program. The *return_code* variable can have one of the following values:

- CM_OK
- CM_PROGRAM_STATE_CHECK
  This return code indicates that the conversation is not in **Confirm**, **Confirm-Send**, or **Confirm-Deallocate** state.
- CM_PROGRAM_PARAMETER_CHECK
  This return code indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

When *return_code* is set to CM_OK:

- The program enters **Receive** state if it received the *status_received* parameter set to CM_CONFIRM_RECEIVED on the preceding Receive call, that is, the conversation was in **Confirm** state.
- The program enters **Send** state if it received the *status_received* parameter set to CM_CONFIRM_SEND_RECEIVED on the preceding Receive call, that is, the conversation was in **Confirm-Send** state.
- The program enters **Reset** state if it received the *status_received* parameter set to CM_CONFIRM_DEALLOC_RECEIVED preceding Receive call, that is, the conversation was in **Confirm-Deallocate** state.

Usage Notes

1. The local program can issue this call only as a reply to a confirmation request; the call cannot be issued at any other time. A confirmation request is generated (by the remote LU) when the remote program makes a call to Confirm. The remote program that has issued Confirm will wait until the local program responds with Confirmed.

2. The program can use this call for various application-level functions. For example, the remote program may send data followed by a confirmation request (using the Confirm call). When the local program receives the confirmation request, it can issue a Confirmed call to indicate that it received and processed the data without error.

Related Information

"Confirm (CMCFM)" on page 52 provides more information on the Confirm call.

"Receive (CMRCV)" on page 74 provides more information on the *status_received* parameter.

"Set_Sync_Level (CMSSL)" on page 116 explains how programs specify the level of synchronization processing.

"Example 4: Validation and Confirmation of Data Reception" on page 32 shows an example program using the Confirmed call.

Deallocate (CMDEAL)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

A program uses the Deallocate (CMDEAL) call to end a conversation. The Deallocate call can include the function of the Flush or Confirm call, depending on the value of the *deallocate_type* conversation characteristic. The *conversation_ID* is no longer assigned when the Deallocation call completes successfully.

Optional set-up:

CALL CMSDT – Set_Deallocate_Type
CALL CMSLD – Set_Log_Data

Format

```
CALL CMDEAL(conversation_ID,
            return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier of the conversation to be ended.

return_code
Specifies the result of the call execution, which is returned to the local program.

If the *deallocate_type* conversation characteristic is set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is set to CM_NONE, or if *deallocate_type* is set to CM_DEALLOCATE_FLUSH or CM_DEALLOCATE_ABEND, the *return_code* variable can have one of the following values:

- CM_OK (deallocation is complete)
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The *deallocate_type* conversation characteristic is set to CM_DEALLOCATE_ABEND, and the conversation is not in **Initialize**, **Send**, **Receive**, **Send-Pending**, **Confirm**, **Confirm-Send**, or **Confirm-Deallocate** state.
  - The *deallocate_type* conversation characteristic is set to CM_DEALLOCATE_SYNC_LEVEL or CM_DEALLOCATE_FLUSH, and the conversation is not in **Send** or **Send-Pending** state.
  - The *deallocate_type* conversation characteristic is set to CM_DEALLOCATE_SYNC_LEVEL or CM_DEALLOCATE_FLUSH; the conversation is basic and in **Send** state; and the program started but did not finish sending a logical record.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PRODUCT_SPECIFIC_ERROR

If the *deallocate_type* conversation characteristic is set to CM_DEALLOCATE_SYNC_LEVEL and the *sync_level* is set to CM_CONFIRM, or if *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, the *return_code* variable can have one of the following values:

- CM_OK (deallocation is complete)
- CM_CONVERSATION_TYPE_MISMATCH
- CM_SECURITY_NOT_VALID
- CM_SYNC_LVL_NOT_SUPPORTED_PGM
- CM_TPN_NOT_RECOGNIZED
- CM_TP_NOT_AVAILABLE_NO_RETRY
- CM_TP_NOT_AVAILABLE_RETRY
- CM_DEALLOCATED_ABEND
- CM_PROGRAM_ERROR_PURGING
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The *deallocate_type* characteristic is set to CM_DEALLOCATE_CONFIRM or CM_DEALLOCATE_SYNC_LEVEL and the conversation is not in **Send** or **Send-Pending** state.
  - The *deallocate_type* characteristic is set to CM_DEALLOCATE_CONFIRM or CM_DEALLOCATE_SYNC_LEVEL; the conversation is basic and in **Send** state; and the program started but did not finish sending a logical record.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

When *return_code* indicates CM_OK, the program enters **Reset** state.

Usage Notes

1. If *deallocate_type* is set to CM_DEALLOCATE_FLUSH or CM_DEALLOCATE_CONFIRM, the execution of Deallocate includes the flushing of the local LU's send buffer.
2. If *deallocate_type* is set to CM_DEALLOCATE_ABEND and the *log_data_length* characteristic is greater than zero, the local LU formats the supplied log data into an Error Log Data GDS variable. After completion of the Deallocate processing, the *log_data* is reset to null and the *log_data_length* is reset to zero.
3. The remote program receives the deallocate notification by means of a *return_code* or *status_received* indication, as follows:
   - CM_DEALLOCATED_NORMAL *return_code*
     This return code indicates that the partner program issued Deallocate with the *sync_level* characteristic set to CM_NONE and *deallocate_type* set to CM_DEALLOCATE_FLUSH or CM_DEALLOCATE_SYNC_LEVEL.

- CM_DEALLOCATED_ABEND *return_code*
  This indicates that the partner program issued Deallocate with *deallocate_type* set to CM_DEALLOCATE_ABEND

  Note: If the remote program has issued Send_Error in **Receive** state, the incoming information containing notice of CM_DEALLOCATED_ABEND is purged and a CM_DEALLOCATED_NORMAL *return_code* is reported instead of CM_DEALLOCATED_ABEND. See "Send_Error (CMSERR)" on page 88 for a complete discussion.

- CM_CONFIRM_DEALLOC_RECEIVED *status_received* indication
  This indicates that the local program issued Deallocate with the *sync_level* set to CM_CONFIRM and *deallocate_type* set to CM_DEALLOCATE_CONFIRM or CM_DEALLOCATE_SYNC_LEVEL.

Related Information

"Example 1: Data Flow in One Direction" on page 23 shows an example program flow using the Deallocate call.

SNA Formats provides a detailed description of GDS variables.

"Set_Log_Data (CMSLD)" on page 102 provides a discussion of the *log_data* characteristic.

"Set_Deallocate_Type (CMSDT)" on page 95 provides a discussion of the *deallocate_type* characteristic and its possible values.

Extract_Conversation_Type (CMECT)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

A program uses the Extract_Conversation_Type (CMECT) call to extract the *conversation_type* characteristic's value for a given conversation. The value is returned in the *conversation_type* parameter.

Format

```
CALL CMECT(conversation_ID,
           conversation_type,
           return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier.

conversation_type
Specifies the conversation type that is returned to the local program. The *conversation_type* can be one of the following:

- CM_BASIC_CONVERSATION
  Indicates that the conversation is allocated as a basic conversation.
- CM_MAPPED_CONVERSATION
  Indicates that the conversation is allocated as a mapped conversation.

Note: If *return_code* is set to CM_PROGRAM_PARAMETER_CHECK, *conversation_type* does not contain a value.

return_code
Specifies the result of the call execution, which is returned to the local program. The *return_code* variable can have one of the following values:

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
  This return code indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

This call does not change the *conversation_type* for the specified conversation.

Related Information

"Set_Conversation_Type (CMSCT)" on page 93 provides more information on the *conversation_type* characteristic.

Extract_Mode_Name (CMEMN)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

A program uses the Extract_Mode_Name (CMEMN) call to extract the *mode_name* characteristic's value for a given conversation. The value is returned to the program in the *mode_name* parameter.

Format

```
CALL CMEMN(conversation_ID,
           mode_name,
           mode_name_length,
           return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier.

mode_name
Specifies the mode name that is returned to the local program. The mode name designates the network properties for the session allocated, or to be allocated, which will carry the conversation specified by the *conversation_ID*.

Note: When *return_code* is set to CM_PROGRAM_PARAMETER_CHECK, *mode_name* does not contain a value.

mode_name_length
Specifies the variable in which is returned the length of the returned *mode_name* parameter.

return_code
Specifies the result of the call execution, which is returned to the local program. The *return_code* variable can have one of the following values:

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

This call does not change the *mode_name* for the specified conversation.

Related Information

"Set_Mode_Name (CMSMN)" on page 104 and "Side Information" on page 11 provide further information on the *mode_name* characteristic.

Extract_Partner_LU_Name (CMEPLN)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

A program uses the Extract_Partner_LU_Name (CMEPLN) call to extract the *partner_LU_name* characteristic's value for a given conversation. The value is returned in the *partner_LU_name* parameter.

Format

```
CALL CMEPLN(conversation_ID,
            partner_LU_name,
            partner_LU_name_length,
            return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier.

partner_LU_name
Specifies the name of the LU where the remote program is located, which is returned to the local program.

Note: If *return_code* is set to CM_PROGRAM_PARAMETER_CHECK, *partner_LU_name* does not contain a value.

partner_LU_name_length
Specifies the variable in which is returned the length of the returned *partner_LU_name* parameter.

return_code
Specifies the result of the call execution, which is returned to the local program. The *return_code* variable can have one of the following values:

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

This call does not change the *partner_LU_name* for the specified conversation.

Related Information

"Set_Partner_LU_Name (CMSPLN)" on page 106 and "Side Information" on page 11 provide more information on the *partner_LU_name* characteristic.

Extract_Sync_Level (CMESL)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

A program uses the Extract_Sync_Level (CMESL) call to extract the *sync_level* characteristic's value for a given conversation. The value is returned to the program in the *sync_level* parameter.

Format

```
CALL CMESL(conversation_ID,
           sync_level,
           return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier.

sync_level
Specifies the *sync_level* characteristic of this conversation, which is returned to the local program. The *sync_level* variable can have one of the following values:

- CM_NONE
  Specifies that the programs will not perform confirmation processing on this conversation. The programs will not issue any calls and will not recognize any returned parameters relating to synchronization.
- CM_CONFIRM
  Specifies that the programs can perform confirmation processing on this conversation. The programs can issue calls and will recognize returned parameters relating to confirmation.

Note: If *return_code* is set to CM_PROGRAM_PARAMETER_CHECK, *sync_level* does not contain a value.

return_code
Specifies the result of the call execution, which is returned to the local program. The *return_code* variable can have one of the following values:

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

This call does not change the *sync_level* for the specified conversation.

Related Information

"Set_Sync_Level (CMSSL)" on page 116 provides more information on the *sync_level* characteristic.

Flush (CMFLUS)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

A program uses the Flush (CMFLUS) call to empty the local LU's send buffer for a given conversation. When notified by CPI Communications that a Flush has been issued, the LU sends any information it has buffered to the remote LU. The information that can be buffered comes from the Allocate, Send_Data, or Send_Error calls. Refer to the descriptions of these calls for more details of when and how buffering occurs.

Format

```
CALL CMFLUS(conversation_ID,
            return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier.

return_code
Specifies the result of the call execution, which is returned to the local program. The *return_code* can be one of the following:

- CM_OK
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is not in **Send** or **Send-Pending** state.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation ID.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

When *return_code* indicates CM_OK:

- The program enters **Send** state if it issues the Flush call while in **Send-Pending** state.
- No state change occurs if the program issues the Flush call while in **Send** state.

Usage Notes

1. This call optimizes processing between the local and remote programs. The local LU normally buffers the data from consecutive Send_Data calls until it has a sufficient amount for transmission. Only then does the local LU transmit the buffered data.

   To avoid this data buffering, the local program can issue a Flush call to minimize any delay in the remote program's processing of the data.

2. The local 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. Contrast the use of Send_Data followed by a call to Flush with the equivalent use of Send_Data after setting *send_type* to CM_SEND_AND_FLUSH.

Related Information

"Set_Send_Type (CMSST)" on page 114 provides a discussion of alternative methods of achieving the Flush function.

"Allocate (CMALLC)" on page 49 provides more information on how information is buffered from the Allocate call.

"Send_Data (CMSEND)" on page 83 provides more information on how information is buffered from the Send_Data call.

"Send_Error (CMSERR)" on page 88 provides more information on how information is buffered from the Send_Error call.

"Data Buffering and Transmission" on page 29 provides a complete discussion of the conditions for data transmission.

"Example 4: Validation and Confirmation of Data Reception" on page 32 shows an example of how a program can use the Flush call to establish a conversation immediately.

Initialize_Conversation (CMINIT)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

A program uses the Initialize_Conversation (CMINIT) call to initialize values for various conversation characteristics before the conversation is allocated (with a call to Allocate). The remote partner program uses the Accept_Conversation call to initialize values for the conversation characteristics on its end of the conversation.

Note: A program can override the values that are initialized by this call using the appropriate Set calls, such as Set_Sync_Level. Once the value is changed, it remains changed until the end of the conversation or until changed again by a Set call.

When the Initialize_Conversation call completes successfully, the following conversation characteristics are initialized:

Conversation Characteristic	Initialized Value
conversation_type	CM_MAPPED_CONVERSATION
deallocate_type	CM_DEALLOCATE_SYNC_LEVEL
error_direction	CM_RECEIVE_ERROR
fill	CM_FILL_LL
log_data	Null value
log_data_length	0
mode_name	Mode name from side information referenced by *sym_dest_name*
mode_name_length	Length of mode name
partner_LU_name	Partner LU name from side information referenced by *sym_dest_name*
partner_LU_name_length	Length of partner LU name
prepare_to_receive_type	CM_PREP_TO_RECEIVE_SYNC_LEVEL
receive_type	CM_RECEIVE_AND_WAIT
return_control	CM_WHEN_SESSION_ALLOCATED
send_type	CM_BUFFER_DATA
sync_level	CM_NONE
TP_name	TP name from side information referenced by *sym_dest_name*
TP_name_length	Length of TP name

Format

```
CALL CMINIT(conversation_ID,
            sym_dest_name
            return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier assigned to the conversation, which is returned to the program. CPI Communications supplies and maintains the *conversation_ID*. If the Initialize_Conversation is successful (*return_code* is set equal to CM_OK), the local program uses the identifier returned in this variable for the rest of the conversation.

sym_dest_name
Specifies the symbolic name of the destination LU and partner program, as well as the mode name for the session on which the conversation is to be carried. The symbolic destination name is provided by the program and points to an entry in the side information table. The appropriate entry in the side information is retrieved and used to initialize the characteristics for the conversation being initialized.

return_code
Specifies the result of the call execution, which is returned to the local program. The *return_code* variable can have one of the following values:

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *sym_dest_name* specifies an unrecognized value.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

When *return_code* indicates CM_OK, the local program enters the **Initialize** state.

Usage Notes

1. For each conversation, CPI Communications assigns a unique identifier, the *conversation_ID*. The program then uses the *conversation_ID* in all future calls intended for that conversation. Initialize_Conversation (or Accept_Conversation, on the opposite side of the conversation), must be issued by the program before any other calls may be made for that conversation.
2. If the program supplies invalid allocation information with the Initialize_Conversation (CMINIT) call, or any subsequent Set calls, the error is detected when the information is processed by Allocate (CMALLC).

Related Information

"Accept_Conversation (CMACCP)" on page 47 provides more information on how conversation characteristics are set by the Accept_Conversation call.

"Allocate (CMALLC)" on page 49 provides more information on how conversation characteristics are set by the Allocate call.

"Conversation Characteristics" on page 14 provides a general overview of conversation characteristics and how they are used by the program and CPI Communications.

"Example 1: Data Flow in One Direction" on page 23 shows an example program flow where Initialize_Conversation is used.

The calls beginning with "Set" and "Extract" in this chapter are used to modify or examine conversation characteristics established by the Initialize_Conversation program call; see the individual call descriptions for details.

"Side Information" on page 11 provides more information on *sym_dest_name*.

Prepare_To_Receive (CMPTR)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

A program uses the Prepare_To_Receive (CMPTR) call to change a conversation from **Send** to **Receive** state in preparation to receive data. This call's function is determined by the value of the *prepare_to_receive_type* conversation characteristic and may include the same function as the Confirm call.

As a result of this call, the local LU's send buffer is flushed.

Optional set-up:

Call CMSPTR – Set_Prepare_To_Receive_Type

Format

```
CALL CMPTR(conversation_ID,
           return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier.

return_code
Specifies the result of the call execution, which is returned to the local program. The *prepare_to_receive_type* currently in effect determines which return codes can be returned to the local program.

If *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_FLUSH, or if *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and the *sync_level* for this conversation is CM_NONE, *return_code* can have one of the following values:

- CM_OK
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The conversation is not in **Send** or **Send-Pending** state.
  - The conversation is basic and in **Send** state, and the program started but did not finish sending a logical record.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PRODUCT_SPECIFIC_ERROR

If *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_CONFIRM, or if *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and the *sync_level* for the conversation is CM_CONFIRM, *return_code* can have one of the following values:

- CM_OK
- CM_CONVERSATION_TYPE_MISMATCH
- CM_SECURITY_NOT_VALID
- CM_SYNC_LVL_NOT_SUPPORTED_PGM
- CM_TPN_NOT_RECOGNIZED
- CM_TP_NOT_AVAILABLE_NO_RETRY
- CM_TP_NOT_AVAILABLE_RETRY
- CM_DEALLOCATED_ABEND
- CM_PROGRAM_ERROR_PURGING
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The conversation is not in **Send** or **Send-Pending** state.
  - The conversation is basic and in **Send** state, and the program started but did not finish sending a logical record.
- CM_PROGRAM_PARAMETER_CHECK
  This return code indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

When *return_code* indicates CM_OK, the program enters the **Receive** state.

Usage Notes

1. If *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_CONFIRM, or if *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* is CM_CONFIRM, the local program regains control when a Confirmed reply is received.
2. The program uses the *prepare_to_receive_type* characteristic set to CM_PREP_TO_RECEIVE_SYNC_LEVEL to transfer send control to the remote program based on one of the following synchronization levels allocated to the conversation:
   - If *sync_level* is set to CM_NONE, send control is transferred to the remote program without any synchronizing acknowledgment.
   - If *sync_level* is set to CM_CONFIRM, send control is transferred to the remote program with confirmation requested.
3. The program uses the *prepare_to_receive_type* characteristic set to CM_PREP_TO_RECEIVE_FLUSH to transfer send control to the remote program without any synchronizing acknowledgment. The *prepare_to_receive_type* characteristic set to CM_PREP_TO_RECEIVE_FLUSH functions the same as the *prepare_to_receive_type* characteristic set to CM_PREP_TO_RECEIVE_SYNC_LEVEL combined with a *sync_level* set to CM_NONE.

4. The program uses the *prepare_to_receive_type* characteristic set to CM_PREP_TO_RECEIVE_CONFIRM to transfer send control to the remote program with confirmation requested. The *prepare_to_receive_type* characteristic set to CM_PREP_TO_RECEIVE_CONFIRM functions the same as the *prepare_to_receive_type* characteristic set to CM_PREP_TO_RECEIVE_SYNC_LEVEL combined with a *sync_level* set to CM_CONFIRM.
5. The remote transaction program receives send control of the conversation by means of the *status_received* parameter, which can have the following values:
   - CM_SEND_RECEIVED
     The local program issued this call with either:
     - *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_FLUSH
       or
     - *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_NONE.
   - CM_CONFIRM_SEND_RECEIVED
     The local program issued this call with either:
     - *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_CONFIRM
       or
     - *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_CONFIRM.
6. After the local program has entered **Receive** state, the remote program enters the corresponding **Send** or **Send-Pending** state when it issues a Receive call and receives send control of the conversation by means of the *status_received* parameter. The remote program can then send data to the local program.

Related Information

"Set_Prepare_To_Receive_Type (CMSPTR)" on page 108 provides more information on the *prepare_to_receive_type* characteristic.

"Set_Sync_Level (CMSSL)" on page 116 provides a discussion of the *sync_level* characteristic and its possible values.

"Example 3: The Sending Program Changes the Data Flow Direction" on page 30 and "Example 5: The Receiving Program Changes the Data Flow Direction" on page 34 show example program flows where the Prepare_To_Receive call is used.

Receive (CMRCV)

TSO/E	CMS	OS/400	QS/2	IMS	CICS
	X				

A program uses the Receive (CMRCV) call to receive information from a given conversation. The information received can be a data record (on a mapped conversation), data (on a basic conversation), conversation status, or a request for confirmation.

Optional set-up:

CALL CMSF (Set_Fill)
CALL CMSRT (Set_Receive_Type)

Format

```
CALL CMRCV(conversation_ID,
           buffer,
           requested_length,
           data_received,
           received_length,
           status_received,
           request_to_send_received,
           return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier.

buffer
Specifies the variable in which the program is to receive the data. If *data_received* is returned to the program with a value of CM_NO_DATA_RECEIVED, *buffer* does not contain anything.

requested_length
Specifies the maximum amount of data the program is to receive.

data_received
Specifies whether or not the program received data, which is returned to the local program.

Note: Unless *return_code* is set to CM_OK or CM_DEALLOCATED_NORMAL, *data_received* does not contain a value.

The *data_received* variable can have one of the following values:

- CM_NO_DATA_RECEIVED (basic and mapped conversations)
  No data is received by the program. Status may be received if the *return_code* is set to CM_OK.
- CM_DATA_RECEIVED (basic conversation only)
  The *fill* characteristic is set to CM_FILL_BUFFER and data (independent of its logical-record format) is received by the program.

- CM_COMPLETE_DATA_RECEIVED (basic and mapped conversations)
  This value indicates one of the following:
  - For mapped conversations, a complete data record or the last remaining portion of the record is received.
  - For basic conversations, *fill* is set to CM_FILL_LL and a complete logical record, or the last remaining portion of the record is received.
- CM_INCOMPLETE_DATA_RECEIVED (basic and mapped conversations)
  This value indicates one of the following:
  - For mapped conversations, less than a complete data record is received.
  - For basic conversations, *fill* is set to CM_FILL_LL, and less than a complete logical record is received.

  Note: For either type of conversation, if *data_received* is set to CM_INCOMPLETE_DATA_RECEIVED, the program must issue another Receive (or possibly multiple Receive calls) to receive the remainder of the data.

received_length
Specifies the variable in which is returned the amount of data the program received, up to the maximum. If the program receives information other than data, *received_length* does not contain a value.

status_received
Specifies the variable in which is returned an indication of whether or not the program received the conversation status.

Note: Unless *return_code* is set to CM_OK, *status_received* does not contain a value.

The *status_received* variable can have one of the following values:

- CM_NO_STATUS_RECEIVED
  No conversation status is received by the program; data may be received.
- CM_SEND_RECEIVED
  The remote program has entered **Receive** state, placing the local program in **Send-Pending** state (if the program also received data on this call) or **Send** state (if the program did not receive data on this call). The local program (who issued the Receive call) can now issue Send_Data.
- CM_CONFIRM_RECEIVED
  The remote program has sent a confirmation request requesting the local program to respond by issuing a Confirmed call. The local program must respond by issuing Confirmed, Send_Error, or Deallocate with *deallocate_type* set to CM_DEALLOCATE_ABEND.
- CM_CONFIRM_SEND_RECEIVED
  The remote program has entered **Receive** state with confirmation requested. The local program must respond by issuing Confirmed, Send_Error, or Deallocate with *deallocate_type* set to CM_DEALLOCATE_ABEND.
- CM_CONFIRM_DEALLOC_RECEIVED
  The remote program has deallocated the conversation with confirmation requested. The local program must respond by issuing Confirmed, Send_Error, or Deallocate with *deallocate_type* set to CM_DEALLOCATE_ABEND.

request_to_send_received
Specifies the variable in which is returned an indication of whether or not the remote program issued a Request_To_Send call.

Note: If *return_code* is set to CM_PROGRAM_PARAMETER_CHECK or CM_PROGRAM_STATE_CHECK, *request_to_send_received* does not contain a value.

The *request_to_send_received* variable can have one of the following values:

- CM_REQ_TO_SEND_RECEIVED
  The local program received a request-to-send notification from the remote program. The remote program issued Request_To_Send, requesting the local program to enter **Receive** state, which places the remote program in **Send** state. See "Request_To_Send (CMRTS)" on page 81 for further discussion of the program's possible responses.
- CM_REQ_TO_SEND_NOT_RECEIVED
  The local program has not received a request-to-send notification.

return_code

Specifies the result of the call execution, which is returned to the local program. The return codes that can be returned depend on the state and characteristics of the conversation at the time this call is issued.

If *receive_type* is set to CM_RECEIVE_AND_WAIT and this call is issued in **Send** state, *return_code* can have one of the following values:

- CM_OK
- CM_CONVERSATION_TYPE_MISMATCH
- CM_SECURITY_NOT_VALID
- CM_SYNC_LVL_NOT_SUPPORTED_PGM
- CM_TPN_NOT_RECOGNIZED
- CM_TP_NOT_AVAILABLE_NO_RETRY
- CM_TP_NOT_AVAILABLE_RETRY
- CM_DEALLOCATED_ABEND
- CM_DEALLOCATED_NORMAL
- CM_PROGRAM_ERROR_NO_TRUNC
- CM_PROGRAM_ERROR_PURGING
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_PRODUCT_SPECIFIC_ERROR

If *receive_type* is set to CM_RECEIVE_AND_WAIT and this call is issued in **Send-Pending** state, *return_code* can be one of the following values:

- CM_OK
- CM_DEALLOCATED_ABEND
- CM_DEALLOCATED_NORMAL
- CM_PROGRAM_ERROR_NO_TRUNC
- CM_PROGRAM_ERROR_PURGING
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_PRODUCT_SPECIFIC_ERROR

If a *receive_type* is set to CM_RECEIVE_AND_WAIT or CM_RECEIVE_IMMEDIATE and this call is issued in **Receive** state, *return_code* can be one of the following:

- CM_OK
- CM_CONVERSATION_TYPE_MISMATCH
- CM_SECURITY_NOT_VALID
- CM_SYNC_LVL_NOT_SUPPORTED_PGM
- CM_TPN_NOT_RECOGNIZED
- CM_TP_NOT_AVAILABLE_NO_RETRY
- CM_TP_NOT_AVAILABLE_RETRY
- CM_DEALLOCATED_ABEND
- CM_DEALLOCATED_NORMAL
- CM_PROGRAM_ERROR_NO_TRUNC
- CM_PROGRAM_ERROR_PURGING
- CM_PROGRAM_ERROR_TRUNC (basic conversation only)
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_PRODUCT_SPECIFIC_ERROR
- CM_UNSUCCESSFUL
  This value indicates that *receive_type* is set to CM_RECEIVE_IMMEDIATE, but there is nothing to receive.

If a state or parameter error has occurred, *return_code* can have one of the following values:

- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The *receive_type* is set to CM_RECEIVE_AND_WAIT and the conversation is not in **Send**, **Send-Pending**, or **Receive** state.
  - The *receive_type* is set to CM_RECEIVE_IMMEDIATE and the conversation is not in **Receive** state.
  - The *receive_type* is set to CM_RECEIVE_AND_WAIT; the conversation is basic and not in **Send** state; and the program started but did not finish sending a logical record.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *requested_length* specifies a value greater than 32767.

State Changes

When *return_code* indicates CM_OK:

- The program enters **Receive** state if a Receive call is issued and all of the following conditions are true:
  - The *receive_type* is set to CM_RECEIVE_AND_WAIT.
  - The conversation is in **Send-Pending** or **Send** state.
  - The *data_received* indicates CM_DATA_RECEIVED, CM_COMPLETE_DATA_RECEIVED, or CM_INCOMPLETE_DATA_RECEIVED.
  - The *status_received* indicates CM_NO_STATUS_RECEIVED.
- The program enters **Send** state when *data_received* is set to CM_NO_DATA_RECEIVED and *status_received* is set to CM_SEND_RECEIVED.

- The program enters **Send-Pending** state when *data_received* is set to CM_DATA_RECEIVED or CM_COMPLETE_DATA_RECEIVED, and *status_received* is set to CM_SEND_RECEIVED.
- The program enters **Confirm**, **Confirm-Send**, or **Confirm-Deallocate** state when *status_received* is set to, respectively, CM_CONFIRM_RECEIVED, CM_CONFIRM_SEND_RECEIVED, or CM_CONFIRM_DEALLOC_RECEIVED.
- No state change occurs when the call is issued in **Receive** state; *data_received* is set to CM_DATA_RECEIVED, CM_COMPLETE_DATA_RECEIVED, or CM_INCOMPLETE_DATA_RECEIVED; and *status_received* indicates CM_NO_STATUS_RECEIVED.

Usage Notes

1. If *receive_type* is set to CM_RECEIVE_AND_WAIT and no data is present when the call is made, CPI Communications waits for information to arrive on the specified conversation before allowing the Receive call to return with the information. If information is already available, the program receives it without waiting.
2. If the program issues Receive call in **Send** state with *receive_type* set to CM_RECEIVE_AND_WAIT, the local LU will flush its send buffer and send all buffered information to the remote program. The local LU will also send a change-of-direction indication. This is a convenient method to change the direction of the conversation, because it leaves the local program in **Receive** state and tells the remote program that it may now begin sending data. The local LU waits for information to arrive.

   Note: A Receive call in **Send** or **Send-Pending** state with a *receive_type* set to CM_RECEIVE_AND_WAIT generates an implicit execution of Prepare_To_Receive with *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_FLUSH followed by a Receive. Refer to "Prepare_To_Receive (CMPTR)" on page 71 for more information.
3. If *receive_type* is set to CM_RECEIVE_IMMEDIATE, a Receive call receives any available information, but does not wait for information to arrive. If information is available, it is returned to the program with an indication of the exact nature of the information received.
4. If the *return_code* indicates CM_PROGRAM_STATE_CHECK or CM_PROGRAM_PARAMETER_CHECK, all other variables will contain no information.
5. A Receive call issued against a mapped conversation can receive only as much of the data record as specified by the *requested_length* parameter. The *data_received* parameter indicates whether the program has received a complete or incomplete data record, as follows:
   - When the program receives a complete data record or the last remaining portion of a data record, the *data_received* parameter is set to CM_COMPLETE_DATA_RECEIVED. The length of the record or portion of the record is less than or equal to the length specified on the *requested_length* parameter.
   - When the program receives a portion of the data record other than the last remaining portion, the *data_received* parameter is set to CM_INCOMPLETE_DATA_RECEIVED. The data record is incomplete because the length of the record is greater than the length specified on the

requested_length parameter. The amount of data received is equal to the length specified.

6. When *fill* is set to CM_FILL_LL on a basic conversation, the program is to receive a logical record, and there are the following possibilities:

   - The program receives a complete logical record or the last remaining portion of a complete record. The length of the record or portion of the record is less than or equal to the length specified on the *requested_length* parameter. The *data_received* parameter is set to CM_COMPLETE_DATA_RECEIVED.
   - The program receives an incomplete logical record for one of the following reasons:
     - The length of the logical record is greater than the length specified on the *requested_length* parameter. In this case, the amount received equals the length specified.
     - Only a portion of the logical record is available because it has been truncated. The portion is equal to or less than the length specified on the *requested_length* parameter.

     The *data_received* parameter is set to CM_INCOMPLETE_DATA_RECEIVED. The program issues another Receive (or possibly multiple Receive calls) to receive the remainder of the logical record.

   Refer to the Send_Data call for a definition of complete and incomplete logical records.

7. When *fill* is set to CM_FILL_BUFFER on a basic conversation, the program is to receive data independent of its logical-record format. The program receives an amount of data equal to or less than the length specified on the *requested_length* parameter. The program can receive less data only when it receives the end of the data. The end of data occurs when it is followed by an indication of a change in the state of the conversation (a change to **Send**, **Send-Pending**, **Confirm**, **Confirm-Send**, **Confirm-Deallocate**, or **Reset** state). The program is responsible for tracking the logical-record format of the data.

8. The Receive call made with *requested_length* set to zero has no special significance. The type of information available is indicated by the *return_code*, *data_received*, and the *status_received* parameters, as usual. If *receive_type* is set to CM_RECEIVE_AND_WAIT and no information is available, this call waits for information to arrive. If *receive_type* is set to CM_RECEIVE_IMMEDIATE, it is possible that no information is available.

   If data is available and *fill* is set to CM_FILL_LL, the *data_received* parameter indicates CM_INCOMPLETE_DATA_RECEIVED. If data is available and *fill* is set to CM_FILL_BUFFER, the *data_received* parameter indicates CM_DATA_RECEIVED. If data is available and the conversation is mapped, the *data_received* parameter is set to COMPLETE_DATA_RECEIVED (that is, 0 bytes are received). In all cases, the program receives no data.

   Note: When *requested_length* is set to zero, receipt of either data or status can be indicated, but not both.

9. The program can receive both data and conversation status on the same call. However, if the remote program truncates a logical record, the local program receives the indication of the truncation on the Receive call issued by the local program after it receives all of the truncated record. The *return_code*, *data_received*, and *status_received* parameters indicate to the program the kind of information the program receives.
10. The request-to-send notification is usually received when the local program is in **Send** state, and reported to the program on a Send_Data call or on a Send_Error call issued in **Send** state. However, the local program can receive the notification while in **Receive** state under the following conditions:
    - When the local program just entered **Receive** state and the remote program issued Request_To_Send before it entered **Send** state.
    - When the remote program just entered **Receive** state with the Prepare_To_Receive call (not Receive with *receive_type* set to CM_RECEIVE_AND_WAIT) and then issued Request_To_Send before the local program enters **Send** state. This can occur because the request-to-send notification is transmitted as an expedited request by the LU. The notification can, therefore, arrive ahead of the request carrying the **Send** indication. Potentially, the local program cannot distinguish this condition from the first. To avoid this ambiguity, the remote program waits until it receives information from the local program before it issues the Request_To_Send.

    Note: The request-to-send notification is returned to the program in addition to (not in place of) the information indicated by the *return_code*, *data_received*, and the *status_received* parameters.

Related Information

"Send_Data (CMSEND)" on page 83 provides more information on complete and incomplete logical records and data records.

"Program Partners and Conversations" on page 9 and "Set_Fill (CMSF)" on page 100 provide more discussion on the use of basic conversations.

All of the example program flows in Chapter 3, "Program-to-Program Communication Tutorial" show programs using the Receive call.

"Request_To_Send (CMRTS)" on page 81 provides a discussion of how a program can place itself into **Receive** state.

"Set_Receive_Type (CMSRT)" on page 110 provides a discussion of the *receive_type* characteristic and its various values.

Request_To_Send (CMRTS)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

The local program uses the Request_To_Send (CMRTS) call to notify the remote program that the local program would like to enter **Send** state for a given conversation. The conversation will be changed to **Send** or **Confirm-Send** state only when the local program subsequently receives, respectively, a CM_SEND_RECEIVED or CM_CONFIRM_SEND_RECEIVED indication from the remote program.

Format

```
CALL CMRTS(conversation_ID,
           return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier.

return_code
Specifies the result of the call execution, which is returned to the local program. The *return_code* variable can have one of the following values:

- CM_OK
- CM_PROGRAM_STATE_CHECK
  This return code indicates that the conversation is not in **Receive**, **Send**, **Send-Pending**, **Confirm**, **Confirm-Send**, or **Confirm-Deallocate** state.
- CM_PROGRAM_PARAMETER_CHECK
  This return code indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. The remote program is informed of the arrival of a request-to-send notification by means of the *request_to_send_received* parameter. The *request_to_send_received* parameter set to CM_REQ_TO_SEND_RECEIVED is a request for the remote program to enter **Receive** state in order to place the partner program (the program that issued the Request_To_Send) in **Send** state.

   A program enters **Receive** state by issuing one of the following calls:

   - The Receive call with *receive_type* set to CM_RECEIVE_AND_WAIT
   - The Prepare_To_Receive call
   - The Send_Data call with *send_type* set to CM_SEND_AND_PREP_TO_RECEIVE.

   After a program issues one of these calls, its partner program is placed into a corresponding **Send** or **Send-Pending** state upon issuing a Receive call. See the *status_received* parameter for the Receive call on page 74 for information about why the state changes from **Receive** to **Send**.

2. The CM_REQ_TO_SEND_RECEIVED value is normally returned to the remote program in the *request_to_send_received* parameter when the remote program is in **Send** state (on a Send_Data or Send_Error call issued in **Send** state). However, the value can also be returned on a Receive call. See "Usage Notes" on the Receive call ("Receive (CMRCV)" on page 74) for more information.
3. When the remote LU receives the request-to-send notification, it retains the notification until the remote program issues a call with the *request_to_send_received* parameter. The remote LU will retain only one request-to-send notification at a time (per conversation). Additional notifications are discarded until the retained notification is indicated to the remote program. Therefore, a local program may issue the Request_To_Send call more times than are indicated to the remote program.

Related Information

"Receive (CMRCV)" on page 74 provides additional information on the *status_received* and *request_to_send_received* parameters.

"Example 5: The Receiving Program Changes the Data Flow Direction" on page 34 shows an example program flow using the Request_To_Send call.

Send_Data (CMSEND)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

A program uses the Send_Data (CMSEND) call to send data to the remote program. When issued during a mapped conversation, this call sends one data record to the remote program. The data record consists entirely of data and is not examined by the LU for possible logical records.

When issued during a basic conversation, this call sends data to the remote program. The data consists of logical records. The amount of data is specified independently of the data format.

Optional set-up:

CALL CMSST – Set_Send_Type

If *send_type* = CM_SEND_AND_PREP_TO_RECEIVE, optional set-up may include:

CALL CMSPTR – Set_Prepare_To_Receive_Type

If *send_type* = CM_SEND_AND_DEALLOCATE, optional set-up may include:

CALL CMSDT – Set_Deallocate_Type

Format

```
CALL CMSEND(conversation_ID,
            buffer,
            send_length,
            request_to_send_received,
            return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier of the conversation.

buffer
When a program issues a Send_Data call during a mapped conversation, *buffer* specifies the data record to be sent. The length of the data record is given by the *send_length* parameter.

When a program issues a Send_Data call during a basic conversation, *buffer* specifies the data to be sent. The data consists of logical records, each containing a 2-byte length field (denoted as LL) followed by a data field. The length of the data field can range from 0 to 32765 bytes. The 2-byte length field contains the following bits:

- A 15-bit binary length of the record
- A high-order bit that is not examined by the LU. It is used, for example, by the LU's mapped conversation component in support of the mapped conversation calls.

The length of the record equals the length of the data field plus the 2-byte length field. Therefore, logical record length values of X'0000', X'0001', X'8000', and X'8001' are not valid.

Note: The logical record length values shown above (such as X'0000') are in the hexadecimal (base-16) numbering system.

send_length

For both basic and mapped conversations, the *send_length* ranges in value from 0 to 32767. It specifies the size of the *buffer* parameter and the number of bytes to be sent on the conversation.

When a program issues a Send_Data call during a mapped conversation and *send_length* is zero, a null data record is sent.

When a program issues a Send_Data call during a basic conversation, *send_length* specifies the size of the *buffer* parameter and is **not** related to the length of a logical record. If *send_length* is zero, no data is sent, and the *buffer* parameter is not important. However, the other parameters and set-up characteristics are significant and retain their meaning as described.

request_to_send_received

Specifies the variable in which is returned an indication of whether or not a request-to-send notification has been received. The *request_to_send_received* variable can have one of the following values:

- CM_REQ_TO_SEND_RECEIVED
  Indicates a request-to-send notification has been received from the remote program. The remote program has issued Request_To_Send, requesting the local program to enter **Receive** state, which places the remote program in **Send** state.
- CM_REQ_TO_SEND_NOT_RECEIVED
  Indicates a request-to-send notification has not been received.

Note: If *return_code* is either CM_PROGRAM_PARAMETER_CHECK or CM_PROGRAM_STATE_CHECK, *request_to_send_received* does not contain a value.

return_code

Specifies the result of the call execution, which is returned to the local program. The *return_code* variable can have one of the following values:

- CM_OK
- CM_CONVERSATION_TYPE_MISMATCH
- CM_SECURITY_NOT_VALID
- CM_SYNC_LVL_NOT_SUPPORTED_PGM
- CM_TPN_NOT_RECOGNIZED
- CM_TP_NOT_AVAILABLE_NO_RETRY
- CM_TP_NOT_AVAILABLE_RETRY
- CM_PROGRAM_ERROR_PURGING
- CM_DEALLOCATED_ABEND
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY

- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The conversation is not in **Send** or **Send-Pending** state.
  - The conversation is basic and in **Send** state; the *send_type* is set to CM_SEND_AND_CONFIRM, CM_SEND_AND_DEALLOCATE, or CM_SEND_AND_PREP_TO_RECEIVE; the *deallocate_type* is not set to CM_DEALLOCATE_ABEND (if *send_type* is set to CM_SEND_AND_DEALLOCATE); and the data does not end on a logical record boundary.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *send_length* is greater than 32767.
  - The *conversation_type* is CM_BASIC_CONVERSATION and *buffer* contains an invalid logical record length (LL) value of X'0000', X'0001', X'8000', or X'8001'.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

When *return_code* indicates CM_OK:

- The program enters **Receive** state when Send_Data is issued with *send_type* set to CM_SEND_AND_PREP_TO_RECEIVE.
- The program enters **Reset** state when Send_Data is issued with *send_type* set to CM_SEND_AND_DEALLOCATE.
- The program enters **Send** state when Send_Data is issued in **Send-Pending** state with *send_type* set to CM_BUFFER_DATA, CM_SEND_AND_FLUSH, or CM_SEND_AND_CONFIRM.
- No state change occurs when Send_Data is issued in **Send** state with *send_type* set to CM_BUFFER_DATA, CM_SEND_AND_FLUSH, or CM_SEND_AND_CONFIRM.

Usage Notes

1. The local LU buffers the data to be sent to the remote LU until it accumulates a sufficient amount of data for transmission (from one or more Send_Data calls), or until the local program issues a call that causes the LU to flush its send buffer. The amount of data sufficient for transmission depends on the characteristics of the session allocated for the conversation, and varies from one session to another.
2. When *request_to_send_received* indicates CM_REQ_TO_SEND_RECEIVED, the remote program is requesting the local program to enter **Receive** state, which places the remote program in **Send** state. See "Request_To_Send (CMRTS)" on page 81 for a discussion of how a program can place itself in **Receive** state.
3. When issued during a mapped conversation, the Send_Data call sends one complete data record. The data record consists entirely of data and CPI Communications does not examine the data for logical record length fields. It is this specification of a complete data record, at send time by the local program and what it sends, that is indicated to the remote program by the *data_received* parameter of the Receive call.

For example, consider a mapped conversation where the local program issues two Send_Data calls with *send_length* set, respectively, to 30, then 50. (These numbers are simplistic for explanatory purposes.) The local program then issues Flush and the 80 bytes of data are sent to the remote LU. The remote program now issues Receive with *requested_length* set to a sufficiently large value, say 1000. The remote program will receive back only 30 bytes of data (indicated by the *received_length* parameter) because this is a complete data record. The completeness of the data record is indicated by the *data_received* variable, which will be set to CM_COMPLETE_DATA_RECEIVED.

The remote program receives the remaining 50 bytes of data (from the second Send) when it performs a second Receive with *requested_length* set to a value greater than or equal to 50.

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

   - One or more complete records, followed by the beginning of a record
   - The end of a record, followed by one or more complete records
   - The end of a record, followed by one or more complete records, followed by the beginning of a record
   - The end of a record, followed by the beginning of a record.

5. The program using a basic conversation must finish sending a logical record before issuing any of the following calls:

   - Confirm
   - Deallocate with *deallocate_type* set to CM_DEALLOCATE_FLUSH, CM_DEALLOCATE_CONFIRM, or CM_DEALLOCATE_SYNC_LEVEL
   - Prepare_To_Receive
   - Receive.

   A program finishes sending a logical record when it sends a complete record or when it truncates an incomplete record. The data must end with the end of a logical record (on a logical record boundary) when Send_data is issued with *send_type* set to CM_SEND_AND_CONFIRM, CM_SEND_AND_DEALLOCATE, or CM_SEND_AND_PREP_TO_RECEIVE.

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

7. During a basic conversation, a program can truncate an incomplete logical record by issuing the Send_Error call. Send_Error causes the LU to flush its send buffer, which includes sending the truncated record. The LU then treats the first two bytes of data specified in the next Send_Data as the LL field. Issuing Send_Data with *send_type* set to CM_SEND_AND_DEALLOCATE and *deallocate_type* set to CM_DEALLOCATE_ABEND, or Deallocate with *deallocate_type* set to CM_DEALLOCATE_ABEND, during a basic conversation also truncates an incomplete logical record.
8. Send_Data is often used in combination with other calls, such as Flush, Confirm, and Prepare_To_Receive. Contrast this usage with the equivalent function available from the use of the Set_Send_Type call prior to issuing a call to Send_Data.

Related Information

"Receive (CMRCV)" on page 74 provides more information on the *data_received* parameter.

"Program Partners and Conversations" on page 9 provides more information on mapped and basic conversations.

SNA Transaction Programmer's Reference Manual for LU Type 6.2 provides further discussion of basic conversations.

"Data Buffering and Transmission" on page 29 provides a complete discussion of controls over data transmission.

All of the example program flows in Chapter 3, "Program-to-Program Communication Tutorial" make use of the Send_Data call.

"Set_Send_Type (CMSST)" on page 114 provides more information on the *send_type* conversation characteristic and the use of it in combination with calls to Send_Data.

Send_Error (CMSERR)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

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

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

Optional set-up:

CALL CMSED – Set_Error_Direction
CALL CMSLD – Set_Log_Data

Format

```
CALL CMSERR(conversation_ID,
            request_to_send_received,
            return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier.

request_to_send_received
Specifies the variable in which is returned an indication of whether or not a request-to-send notification has been received. The *request_to_send_received* variable can have one of the following values:

- CM_REQ_TO_SEND_RECEIVED
  The remote program issued a Request_To_Send call requesting the local program to enter **Receive** state, which places the remote program in **Send** state.
- CM_REQ_TO_SEND_NOT_RECEIVED
  A request-to-send notification has not been received.

Note: If *return_code* is set to either CM_PROGRAM_PARAMETER_CHECK or CM_PROGRAM_STATE_CHECK, *request_to_send_received* does not contain a value.

return_code
Specifies the result of the call execution, which is returned to the local program. The value for *return_code* depends on the state of the conversation at the time this call is issued.

If the Send_Error is issued in **Send** state, *return_code* can have one of the following values:

- CM_OK
- CM_CONVERSATION_TYPE_MISMATCH
- CM_SECURITY_NOT_VALID
- CM_SYNC_LVL_NOT_SUPPORTED_PGM
- CM_TPN_NOT_RECOGNIZED
- CM_TP_NOT_AVAILABLE_NO_RETRY
- CM_TP_NOT_AVAILABLE_RETRY
- CM_DEALLOCATED_ABEND
- CM_PROGRAM_ERROR_PURGING
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_PRODUCT_SPECIFIC_ERROR

If the Send_Error is issued in **Receive** state, *return_code* can have one of the following values:

- CM_OK
- CM_DEALLOCATED_NORMAL
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_PRODUCT_SPECIFIC_ERROR

If the Send_Error is issued in **Send-Pending**, **Confirm**, **Confirm-Send**, or **Confirm-Deallocate** state, *return_code* can have one of the following values:

- CM_OK
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_PRODUCT_SPECIFIC_ERROR

Otherwise, the conversation is in **Reset** or **Initialize** state and *return_code* has one of the following values:

- CM_PROGRAM_PARAMETER_CHECK
  - The *conversation_ID* specifies an unassigned identifier.
- CM_PROGRAM_STATE_CHECK

State Changes

When *return_code* indicates CM_OK:

- The program enters **Send** state when the call is issued in **Receive**, **Confirm**, **Confirm-Send**, **Confirm-Deallocate**, or **Send-Pending** state.
- No state change occurs when the call is issued in **Send** state.

Usage Notes

1. The LU can send the error notification to the remote LU immediately (during the processing of this call), or the LU can delay sending the notification until a later time. If the LU delays sending the notification, it buffers the notification until it has accumulated a sufficient amount of information for transmission, or until the local program issues a call that causes the LU to flush its send buffer.

2. The amount of information sufficient for transmission depends on the characteristics of the session allocated for the conversation, and varies from one session to another. Transmission of the information can begin immediately if the *log_data* characteristic has been specified with sufficient log data, or transmission can be delayed until sufficient data from subsequent Send_Data calls is also buffered.
3. To make sure that the remote program receives the error notification as soon as possible, the local program can issue Flush immediately after Send_Error.
4. The issuance of Send_Error is reported to the remote program as one of the following return codes:

   - CM_PROGRAM_ERROR_TRUNC (basic conversation)
     The local program issued Send_Error in **Send** state after sending an incomplete logical record (see "Send_Data (CMSEND)" on page 83). The record has been truncated.
   - CM_PROGRAM_ERROR_NO_TRUNC (basic and mapped conversations)
     The local program issued Send_Error in **Send** state after sending a complete logical record (basic) or data record (mapped); or before sending any record; or the local program issued Send_Error in **Send-Pending** state with *error_direction* set to CM_SEND_ERROR. No truncation has occurred.
   - CM_PROGRAM_ERROR_PURGING (basic and mapped conversations)
     The local program issued Send_Error in **Receive** state. All information sent by the remote program and not yet received by the local program has been purged, or the local program issued Send_Error in **Send-Pending** state with *error_direction* set to CM_RECEIVE_ERROR or in **Confirm**, **Confirm-Send**, or **Confirm-Deallocate** state, in which case no purging has occurred.
5. When Send_Error is issued in **Receive** state, incoming information is also purged. Because of this purging, the *return_code* of CM_DEALLOCATED_NORMAL is reported instead of:

   - CM_ALLOCATE_FAILURE_NO_RETRY
   - CM_ALLOCATE_FAILURE_RETRY
   - CM_CONVERSATION_TYPE_MISMATCH
   - CM_SECURITY_NOT_VALID
   - CM_SYNC_LVL_NOT_SUPPORTED_PGM
   - CM_TPN_NOT_RECOGNIZED
   - CM_TP_NOT_AVAILABLE_NO_RETRY
   - CM_TP_NOT_AVAILABLE_RETRY
   - CM_DEALLOCATED_ABEND

   Similarly, a return code of CM_OK is reported instead of:

   - CM_PROGRAM_ERROR_NO_TRUNC
   - CM_PROGRAM_ERROR_PURGING
   - CM_PROGRAM_ERROR_TRUNC (basic conversation only)

   The following types of incoming information are also purged:

   - Data sent with the Send_Data call.
   - Confirmation request sent with the Send_Data, Confirm, Prepare_To_Receive, or Deallocate calls.

If the confirmation request was sent with *deallocate_type* set to CM_DEALLOCATE_CONFIRM or CM_DEALLOCATE_SYNC_LEVEL, the deallocation request will also be purged.

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

6. The program can use this call for various application-level functions. For example, the program can issue this call to truncate an incomplete logical record it is sending; to inform the remote program of an error detected in data received; or to reject a confirmation request.

7. If the *log_data_length* characteristic is greater than zero, the LU formats the supplied log data into an Error Log Data GDS variable. The data supplied by the program is any data the program wants to have logged. The data is placed in the message text portion of the Error Log GDS variable created by the LU. The LU formats the GDS variable, filling in the appropriate length fields and the product set ID portion of the GDS variable. After completion of the Send_Error processing, *log_data* is reset to null, and *log_data_length* is reset to zero.

8. The *error_direction* characteristic is significant only when Send_Error is issued during a conversation in **Send-Pending** state (that is, the Send_Error is issued immediately following a Receive on which both data and a *status_received* parameter set to CM_SEND_RECEIVED is received). In this case, Send_Error could be reporting one of the following types of errors:

   - An error in the received data (in the receive flow)
   - An error having nothing to do with the received data, but instead being the result of processing performed by the program after it had successfully received and processed the data (in the send flow).

   Because the LU cannot tell which of the two errors occurred, the program has to supply the *error_direction* information.

   The default for *error_direction* is CM_RECEIVE_ERROR. A program can override the default using the Set_Error_Direction call before issuing Send_Error.

   Once changed, the new *error_direction* value remains in effect until the program changes it again. Therefore, a program should issue Set_Error_Direction before issuing Send_Error in **Send-Pending** state.

   If the program is not in **Send-Pending** state, the *error_direction* characteristic is ignored.

Related Information

"Example 6: Reporting Errors" on page 36 and "Example 7: Error Direction and Send-Pending State" on page 38 provide example program flows using Send_Error and the **Send-Pending** state; "Set_Error_Direction (CMSED)" on page 98 provides further information on the *error_direction* characteristic.

"Usage Notes" of "Request_To_Send (CMRTS)" on page 81 provides more information on how a program enters **Receive** state.

SNA Formats provides a detailed description of GDS variables.

"Send_Data (CMSEND)" on page 83 provides a discussion of basic conversations and logical records.

"Set_Log_Data (CMSLD)" on page 102 provides a description of the *log_data* characteristic.

Set_Conversation_Type (CMSCT)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

Set_Conversation_Type (CMSCT) is used by a program to set the *conversation_type* characteristic for a given conversation. It overrides the value assigned with the Initialize_Conversation call.

Note: A program cannot use Set_Conversation_Type after an Allocate has been issued. Only the program that initiates the conversation (using the Initialize_Conversation call) can issue the Set_Conversation call.

Format

```
CALL CMSCT(conversation_ID,
           conversation_type,
           return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier.

conversation_type
Specifies the type of conversation to be allocated when Allocate is issued. The *conversation_type* variable can have one of the following values:

- CM_BASIC_CONVERSATION
  Specifies the allocation of a basic conversation.
- CM_MAPPED_CONVERSATION
  Specifies the allocation of a mapped conversation.

return_code
Specifies the result of the call execution, which is returned to the local program. The *return_code* variable can have one of the following values:

- CM_OK
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is not in **Initialize** state.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *conversation_type* specifies an undefined value.
  - The *conversation_type* is set to CM_MAPPED_CONVERSATION, but *fill* is set to CM_FILL_BUFFER.
  - The *conversation_type* is set to CM_MAPPED_CONVERSATION, but a prior call to Set_Log_Data is still in effect.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. Some calls, such as Set_Fill, can only be issued against a basic conversation. However, a program may attempt to issue the following incorrect sequence of calls:

   a. Initialize_Conversation, setting *conversation_type* to CM_MAPPED_CONVERSATION

   b. Set_Conversation_Type, setting *conversation_type* to CM_BASIC_CONVERSATION

   c. Set_Fill, setting *fill* to CM_FILL_BUFFER

   d. Set_Conversation_Type setting *conversation_type* to CM_MAPPED_CONVERSATION

   e. Allocate

   In the above sequence, the program attempts to set *conversation_type* to CM_MAPPED_CONVERSATION after the *fill* characteristic has been set with *conversation_type* (CM_BASIC_CONVERSATION) in effect. An error will be returned by Set_Conversation_Type because *conversation_type* can no longer be changed.

2. Because of the detailed manipulation of the data and resulting complexity of error conditions, the use of basic conversations should be regarded as intended for advanced programmers.

3. If a *return_code* other than CM_OK is returned on the call, the *conversation_type* conversation characteristic is unchanged.

Related Information

"Program Partners and Conversations" on page 9 and the "Usage Notes" section of "Send_Data (CMSEND)" on page 83 provide more information on the differences between mapped and basic conversations.

Set_Deallocate_Type (CMSDT)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

Set_Deallocate_Type (CMSDT) is used by a program to set the *deallocate_type* characteristic for a given conversation. Set_Deallocate_Type overrides the value that was assigned when either the Initialize_Conversation or the Accept_Conversation call was issued.

Format

```
CALL CMSDT(conversation_ID,
           deallocate_type,
           return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier.

deallocate_type
Specifies the type of deallocation to be performed. The *deallocate_type* variable can have one of the following values:

- CM_DEALLOCATE_SYNC_LEVEL
  Perform deallocation based on the *sync_level* characteristic in effect for this conversation:
  - If *sync_level* is set to CM_NONE, execute the function of the Flush call and deallocate the conversation normally.
  - If *sync_level* is set to CM_CONFIRM, execute the function of the Confirm call and if successful (as indicated by a return code of CM_OK on the Deallocate call, or a return code of CM_OK on the Send_Data call with *send_type* set to CM_SEND_AND_DEALLOCATE), deallocate the conversation normally. If the Confirm call is not successful, the state of the conversation is determined by the return code.
- CM_DEALLOCATE_FLUSH
  Execute the function of the Flush call and deallocate the conversation normally.
- CM_DEALLOCATE_CONFIRM
  Execute the function of the Confirm call and if successful (as indicated by a return code of CM_OK on the Deallocate call, or a return code of CM_OK on the Send_Data call with *send_type* set to CM_SEND_AND_DEALLOCATE), deallocate the conversation normally. If the Confirm is not successful, the state of the conversation is determined by the return code.
- CM_DEALLOCATE_ABEND
  Execute the function of the Flush call when the conversation is in **Send** state and deallocate the conversation abnormally. Data purging can occur when the conversation is in **Receive** state. If the conversation is a basic conversation, logical-record truncation can occur when the conversation is in **Send** state.

return_code
Specifies the result of the call execution, which is returned to the local program. The *return_code* variable can have one of the following values:

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, and the conversation is assigned with *sync_level* set to CM_NONE.
  - The *deallocate_type* specifies an undefined value.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. A *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL is used by a program to deallocate a conversation based on the conversation's synchronization level:
   - If *sync_level* is set to CM_NONE, the conversation is unconditionally deallocated.
   - If *sync_level* is set to CM_CONFIRM, the conversation is deallocated when the remote program responds to the confirmation request by issuing the Confirmed call. The conversation remains allocated when the remote program responds to the confirmation request by issuing the Send_Error call.
2. A *deallocate_type* set to CM_DEALLOCATE_FLUSH is used by a program to unconditionally deallocate the conversation regardless of the level of synchronization. The *deallocate_type* set to CM_DEALLOCATE_FLUSH is functionally equivalent to *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL combined with a *sync_level* set to CM_NONE.
3. A *deallocate_type* set to CM_DEALLOCATE_CONFIRM is used by a program to conditionally deallocate the conversation, depending on the remote program's response, when the *sync_level* is set to CM_CONFIRM. The *deallocate_type* set to CM_DEALLOCATE_CONFIRM is functionally equivalent to *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL combined with a *sync_level* set to CM_CONFIRM.

   The conversation is deallocated when the remote program responds to the confirmation request by issuing Confirmed. The conversation remains allocated when the remote program responds to the confirmation request by issuing Send_Error.
4. A *deallocate_type* set to CM_DEALLOCATE_ABEND is used by a program to unconditionally deallocate a conversation regardless of its synchronization level and its current state. Specifically, the parameter is used when the program detects an error condition that prevents further useful communications (communications that would lead to successful completion of the transaction).
5. If a *return_code* other than CM_OK is returned on the call, the *deallocate_type* conversation characteristic is unchanged.

Related Information

"Deallocate (CMDEAL)" on page 56 provides further discussion on the use of the *deallocate_type* characteristic in the deallocation of a conversation.

"Set_Sync_Level (CMSSL)" on page 116 provides information on how the *sync_level* characteristic is used in combination with the *deallocate_type* characteristic in the deallocation of a conversation.

Set_Error_Direction (CMSED)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

Set_Error_Direction (CMSED) is used by a program to set the *error_direction* characteristic for a given conversation. Set_Error_Direction overrides the value that was assigned when the Initialize_Conversation or the Accept_Conversation calls were issued.

Format

```
CALL CMSED(conversation_ID,
           error_direction,
           return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier.

error_direction
Specifies the direction of the data flow in which the program detected an error. This parameter is significant only if Send_Error is issued in **Send-Pending** state (that is, immediately after a Receive on which both data and a conversation status of CM_SEND_RECEIVED are received). Otherwise, the *error_direction* value is ignored when the program issues Send_Error.

The *error_direction* variable can have one of the following values:

- CM_RECEIVE_ERROR
  Specifies that the program detected an error in the data it received from the remote program.
- CM_SEND_ERROR
  Specifies that the program detected an error while preparing to send data to the remote program.

return_code
Specifies the result of the call execution, which is returned to the local program. The *return_code* variable can have one of the following values:

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *error_direction* specifies an undefined value.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. The *error_direction* parameter is significant only if Send_Error is issued immediately after a Receive on which both data and a conversation status of CM_SEND_RECEIVED are received (when the conversation is in **Send-Pending** state). Otherwise, the *error_direction* value is ignored when the program issues Send_Error. In this situation, the Send_Error may result from one of the following errors:

   - An error in the received data (in the receive flow)
   - An error having nothing to do with the received data, but instead being the result of processing performed by the program after it had successfully received and processed the data (in the send flow).

   Because the LU in this situation cannot tell which error occurred, the program has to supply the *error_direction* information.

   The *error_direction* defaults to a value of CM_RECEIVE_ERROR. To override the default, a program can issue the Set_Error_Direction call prior to issuing Send_Error.

   Once changed, the new *error_direction* value remains in effect until the program changes it again. Therefore, a program should issue Set_Error_Direction before issuing Send_Error in **Send-Pending** state.

   If the program is not in **Send-Pending** state, the *error_direction* characteristic is ignored.

2. If a *return_code* other than CM_OK is returned on the call, the *error_direction* conversation characteristic is unchanged.

Related Information

"Example 7: Error Direction and Send-Pending State" on page 38 provides an example program using Set_Error_Direction.

"Send_Error (CMSERR)" on page 88 provides more information on reporting errors.

Set_Fill (CMSF)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

Set_Fill (CMSF) is used by a program to set the *fill* characteristic for a given conversation. Set_Fill overrides the value that was assigned by the Initialize_Conversation or Accept_Conversation calls.

Note: This call applies only to basic conversations. The *fill* characteristic is ignored for mapped conversations.

Format

```
CALL CMSF(conversation_ID,
          fill,
          return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier.

fill
Specifies whether the program is to receive data in terms of the logical-record format of the data. The *fill* variable can have one of the following values:

- CM_FILL_LL
  Specifies that the program is to receive one complete or truncated logical record, or a portion of the logical record that is equal to the length specified by the *requested_length* parameter of the Receive call.
- CM_FILL_BUFFER
  Specifies that the program is to receive data independent of its logical-record format. The amount of data received will be equal to or less than the length specified by the *requested_length* parameter of the Receive call. The amount is less than the requested length when the program receives the end of the data.

return_code
Specifies the result of the call execution, which is returned to the local program. The *return_code* variable can have one of the following values:

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *conversation_type* specifies CM_MAPPED_CONVERSATION.
  - The *fill* characteristic specifies an undefined value.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This program does not cause a state change.

Usage Notes

1. The *fill* value provided (for a basic conversation) is used on all subsequent Receive calls for the specified conversation until changed by the program with another Set_Fill call.
2. If a *return_code* other than CM_OK is returned on the call, the *fill* conversation characteristic is unchanged.

Related Information

"Receive (CMRCV)" on page 74 provides more information on how the *fill* characteristic is used for basic conversations.

Set_Log_Data (CMSLD)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

Set_Log_Data (CMSLD) is used by a program to set the *log_data* and *log_data_length* characteristics for a given conversation. Set_Log_Data overrides the values that were assigned with the Initialize_Conversation or Accept_Conversation calls.

Note: This call applies only to basic conversations. The *log_data* characteristic is ignored for mapped conversations.

Format

```
CALL CMSLD(conversation_ID,
           log_data,
           log_data_length,
           return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier.

log_data
Specifies the program-unique error information that is to be placed in the system error logs of the LUs supporting this conversation. The error information supplied is formatted by the sending LU into an Error Log GDS variable. The data supplied by the program is any data the program wants to have logged. The data is placed in the message text portion of the Error Log GDS variable created by the LU. The LU formats the GDS variable, filling in the appropriate length fields and the product-set ID portion of the GDS variable.

log_data_length
Specifies the length of the product-unique error information. The length can be from 0 to 512 bytes. If zero, no log data is sent for this call and the *log_data* parameter is not significant.

return_code
Specifies the result of the call execution, which is returned to the local program. The *return_code* variable can have one of the following values:

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
  This value can be one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *conversation_type* is set to CM_MAPPED_CONVERSATION.
  - The *log_data_length* specifies a value greater than 512 or less than 0.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. The LU resets the *log_data* and *log_data_length* characteristics to their initial (null) values after an issuance of Send_Error or Deallocate (*deallocate_type* set to CM_DEALLOCATE_ABEND) calls. Therefore, the *log_data* is sent to the remote LU only once per issuance of an error indication.
2. If a *return_code* other than CM_OK is returned on the call, the *log_data* and *log_data_length* conversation characteristics are unchanged.

Related Information

SNA Formats provides a detailed description of GDS variables.

"Send_Error (CMSERR)" on page 88 and "Deallocate (CMDEAL)" on page 56 provide further discussion on how the *log_data* characteristic is used.

Set_Mode_Name (CMSMN)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

Set_Mode_Name (CMSMN) is used by a program to set the *mode_name* and *mode_name_length* characteristic for a conversation. Set_Mode_Name overrides the system-defined value, which was originally acquired from the side information using the *sym_dest_name*.

Issuing this call does not change the values in the side information. It only changes the *mode_name* for this conversation.

Note: A program cannot issue the Set_Mode_Name call after an Allocate is issued. Only the program that initiates the conversation (using the Initialize_Conversation call) can issue this call.

Format

```
CALL CMSMN(conversation_ID,
           mode_name,
           mode_name_length,
           return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier.

mode_name
Specifies the mode name designating the network properties for the session to be allocated for the conversation. The network properties include, for example, the class of service to be used, and whether data is to be enciphered.

Note: Only an SNA service transaction program can specify the SNA-defined mode name SNASVCMG.

mode_name_length
Specifies the length of the mode name. The length can be from zero to eight bytes.

return_code
Specifies the result of the call execution, which is returned to the local program. The *return_code* variable can have one of the following values:

- CM_OK
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is not in **Initialize** state.

- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *mode_name_length* specifies a value less than zero or greater than eight.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. Specification of a mode name that is not recognized by the LU is not detected on this call. It is detected on the subsequent Allocate call.
2. If a *return_code* other than CM_OK is returned on the call, the *mode_name* and *mode_name_length* conversation characteristics are unchanged.

Related Information

"SNA Service Transaction Programs" on page 149 provides a discussion of SNA service transaction programs.

"Side Information" on page 11 provides further discussion of the *mode_name* conversation characteristic.

Set_Partner_LU_Name (CMSPLN)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

Set_Partner_LU_Name (CMSPLN) is used by a program to set the *partner_LU_name* characteristic for a conversation. Set_Partner_LU_Name overrides the current value for the *partner_LU_Name*, which was originally acquired from the side information using the *sym_dest_name*.

Issuing this call does not change the information in the side information. It only changes the *partner_LU_Name* and *partner_LU_Name_length* for this conversation.

Note: A program cannot issue Set_Partner_LU_Name after an Allocate call is issued. Only the program that initiated the conversation (issued the Initialize_Conversation) can issue Set_Partner_LU_Name.

Format

```
CALL CMSPLN(conversation_ID,
            partner_LU_name,
            partner_LU_name_length,
            return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier.

partner_LU_name
Specifies the name of the remote LU at which the remote program is located. This LU name is any name by which the local LU knows the remote LU for purposes of allocating a conversation.

partner_LU_name_length
Specifies the length of the partner LU name. The length can be from 1 to 17 bytes.

return_code
Specifies the result of the call execution, which is returned to the local program. The *return_code* variable can have one of the following values:

- CM_OK
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is not in **Initialize** state.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *partner_LU_name_length* is set to a value less than 1 or greater than 17.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This program does not cause a state change.

Usage Notes

If a *return_code* other than CM_OK is returned on the call, the *partner_LU_name* and *partner_LU_name_length* conversation characteristics are unchanged.

Related Information

"Side Information" on page 11 and note 2 of Table 6 on page 129 provide further discussion of the *partner_LU_name* conversation characteristic.

Set_Prepare_To_Receive_Type (CMSPTR)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

Set_Prepare_To_Receive_Type (CMSPTR) is used by a program to set the *prepare_to_receive_type* characteristic for a conversation. This call overrides the value that was assigned by the Initialize_Conversation or Accept_Conversation calls.

Format

```
CALL CMSPTR(conversation_ID,
            prepare_to_receive_type,
            return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier.

prepare_to_receive_type
Specifies the type of prepare-to-receive to be performed for this conversation. The *prepare_to_receive_type* variable can have one of the following values:

- CM_PREP_TO_RECEIVE_SYNC_LEVEL
  Perform the prepare-to-receive based on one of the following *sync_level* settings:
  - If *sync_level* is set to CM_NONE, execute the function of the Flush call and enter **Receive** state.
  - If *sync_level* is set to CM_CONFIRM, execute the function of the Confirm call and if successful (as indicated by a return code of CM_OK on the Prepare_To_Receive call, or a return code of CM_OK on the Send_Data call with *send_type* set to CM_SEND_AND_PREP_TO_RECEIVE), enter **Receive** state. If Confirm is not successful, the state of the conversation is determined by the return code.
- CM_PREP_TO_RECEIVE_FLUSH
  Execute the function of the Flush call and enter **Receive** state.
- CM_PREP_TO_RECEIVE_CONFIRM
  Execute the function of the Confirm call and if successful (as indicated by a return code of CM_OK on the Prepare_To_Receive call, or a return code of CM_OK on the Send_Data call with *send_type* set to CM_SEND_AND_PREP_TO_RECEIVE), enter **Receive** state. If it is not successful, the state of the conversation is determined by the return code.

Note: The execution of the Flush or Confirm function as part of the Prepare_To_Receive call includes the flushing of the LU's send buffer.

return_code
Specifies the result of the call execution, which is returned to the local program. The *return_code* variable can have one of the following values:

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *prepare_to_receive_type* is CM_PREP_TO_RECEIVE_CONFIRM, but the conversation is assigned with *sync_level* set to CM_NONE.
  - The *prepare_to_receive_type* is set to an undefined value.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

If a *return_code* other than CM_OK is returned on the call, the *prepare_to_receive_type* conversation characteristic is unchanged.

Related Information

"Prepare_To_Receive (CMPTR)" on page 71 provides a discussion of how the *prepare_to_receive_type* is used.

"Example 5: The Receiving Program Changes the Data Flow Direction" on page 34 shows an example program using the Prepare_To_Receive call.

Set_Receive_Type (CMSRT)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

Set_Receive_Type (CMSRT) is used by a program to set the *receive_type* characteristic for a conversation. Set_Receive_Type overrides the value that was assigned by the Initialize_Conversation or Accept_Conversation calls.

Format

```
CALL CMSRT(conversation_ID,
           receive_type,
           return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier.

receive_type
Specifies the type of receive to be performed. The *receive_type* variable can have one of the following values:

- CM_RECEIVE_AND_WAIT
  The Receive call is to wait for information to arrive on the specified conversation. If information is already available, the program receives it without waiting.
- CM_RECEIVE_IMMEDIATE
  The Receive call is to receive any information that is available from the specified conversation, but is not to wait for information to arrive.

return_code
Specifies the result of the call execution, which is returned to the local program. The *return_code* variable can have one of the following values:

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *receive_type* specifies an undefined value.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

If a *return_code* other than CM_OK is returned on the call, the *receive_type* conversation characteristic is unchanged.

Related Information

"Receive (CMRCV)" on page 74 provides a discussion of how the *receive_type* characteristic is used.

"Example 3: The Sending Program Changes the Data Flow Direction" on page 30 provides a discussion of how a program can use Set_Receive_Type with a value of CM_RECEIVE_IMMEDIATE.

Set_Return_Control (CMSRC)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

Set_Return_Control (CMSRC) is used to set the *return_control* characteristic for a given conversation. Set_Return_Control overrides the value that was assigned with the Initialize_Conversation call.

Note: A program cannot issue the Set_Return_Control after an Allocate has been issued for a conversation. Only the program that initiates the conversation (with the Initialize_Conversation) can issue this call.

Format

```
CALL CMSRC(conversation_ID,
           return_control,
           return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier.

return_control
Specifies when a program will receive control back after issuing a call to Allocate. The *return_control* can have one of the following values:

- CM_WHEN_SESSION_ALLOCATED
  Allocate a session for the conversation before returning control to the program.
- CM_IMMEDIATE
  Allocate a session for the conversation if a session is immediately available and return control to the program with one of the following return codes indicating whether or not a session is allocated.
  - A return code of CM_OK indicates a session was immediately available and has been allocated for the conversation. A session is immediately available when it is active; the session is not allocated to another conversation; and the local LU is the contention winner for the session.
  - A return code of CM_UNSUCCESSFUL indicates a session is not immediately available. Allocation is not performed.

return_code
Specifies the result of the call execution, which is returned to the local program. The *return_code* variable can have one of the following values:

- CM_OK
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is not in **Initialize** state.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *return_control* specifies an undefined value.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. An allocation error resulting from the local LU's failure to obtain a session for the conversation is reported on the Allocate call. An allocation error resulting from the remote LU's rejection of the allocation request is reported on a subsequent conversation call.
2. Two LUs connected by a session may both attempt to allocate a conversation on the session at the same time. This is called contention. Contention is resolved by making one LU the contention winner of the session and the other LU the contention loser of the session. The contention-winner LU allocates a conversation on a session without asking permission from the contention-loser LU. Conversely, the contention-loser LU requests permission from the contention-winner LU to allocate a conversation on the session, and the contention-winner LU either grants or rejects the request. For further information, see *SNA Transaction Programmer's Reference Manual for LU Type 6.2*.

   Contention may result in an CM_UNSUCCESSFUL return code for programs specifying CM_IMMEDIATE.
3. If a *return_code* other than CM_OK is returned on the call, the *return_control* conversation characteristic is unchanged.

Related Information

"Allocate (CMALLC)" on page 49 provides more discussion on the use of the *return_control* characteristic in allocating a conversation.

Set_Send_Type (CMSST)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

Set_Send_Type (CMSST) is used by a program to set the *send_type* characteristic for a conversation. Set_Send_Type overrides the value that was assigned with the Initialize_Conversation or Accept_Conversation calls.

Format

```
CALL CMSST(conversation_ID,
           send_type,
           return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier.

send_type
Specifies what, if any, information is to be sent to the remote program in addition to the data supplied on the Send_Data call, and whether the data is to be sent immediately or buffered.

The *send_type* variable can have one of the following values:

- CM_BUFFER_DATA
  No additional information is to be sent to the remote program. Further, the supplied data may not be sent immediately but, instead, may be buffered until a sufficient quantity is accumulated.
- CM_SEND_AND_FLUSH
  No additional information is to be sent to the remote program. However, the supplied data is sent immediately rather than buffered. Send_Data with *send_type* set to CM_SEND_AND_FLUSH is functionally equivalent to a Send_Data with *send_type* set to CM_BUFFER_DATA followed by a Flush call.
- CM_SEND_AND_CONFIRM
  The supplied data is to be sent to the remote program immediately, along with a request for confirmation. Send_Data with *send_type* set to CM_SEND_AND_CONFIRM is functionally equivalent to Send_Data with *send_type* set to CM_BUFFER_DATA followed by a Confirm call.
- CM_SEND_AND_PREP_TO_RECEIVE
  The supplied data is to be sent to the remote program immediately, along with send control of the conversation. Send_Data with *send_type* set to CM_SEND_AND_PREP_TO_RECEIVE is functionally equivalent to Send_Data with *send_type* set to CM_BUFFER_DATA followed by a Prepare_To_Receive call.
- CM_SEND_AND_DEALLOCATE
  The supplied data is to be sent to the remote program immediately, along with a deallocation notification. Send_Data with *send_type* set to CM_SEND_AND_DEALLOCATE is functionally equivalent to Send_Data with *send_type* set to CM_BUFFER_DATA followed by a call to Deallocate.

return_code

Specifies the result of the call execution, which is returned to the local program. The *return_code* variable can have one of the following values:

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *send_type* is set to CM_SEND_AND_CONFIRM and the conversation is assigned with *sync_level* set to CM_NONE.
  - The *send_type* specifies an undefined value.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

If a *return_code* other than CM_OK is returned on the call, the *send_type* conversation characteristic is unchanged.

Related Information

"Send_Data (CMSEND)" on page 83 provides a discussion of how the *send_type* characteristic is used by Send_Data.

The same function of a call to Send_Data with different values of the *send_type* conversation characteristic in effect can be achieved by combining Send_Data with other calls:

- "Flush (CMFLUS)" on page 66
- "Confirm (CMCFM)" on page 52
- "Prepare_To_Receive (CMPTR)" on page 71
- "Deallocate (CMDEAL)" on page 56

"Example 5: The Receiving Program Changes the Data Flow Direction" on page 34 shows an example program flow using the Set_Send_Type call.

Set_Sync_Level (CMSSL)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

Set_Sync_Level (CMSSL) is used by a program to set the *sync_level* characteristic for a given conversation. The *sync_level* characteristic is used to specify the level of synchronization processing between the two programs. It determines whether or not the programs support the use of Confirm and Confirmed calls. Set_Sync_Level overrides the value that was assigned with the Initialize_Conversation call.

Note: A program cannot use the Set_Sync_Level call after an Allocate has been issued. Only the program that initiates a conversation (using the Initialize_Conversation call) can issue this call.

Format

```
CALL CMSSL(conversation_ID,
           sync_level,
           return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier.

sync_level
Specifies the synchronization level that the local and remote programs can use on this conversation. The *sync_level* can have one of the following values:

- CM_NONE
  The programs will not perform confirmation processing on this conversation. The programs will not issue any calls and will not recognize any returned parameters relating to synchronization.
- CM_CONFIRM
  The programs can perform confirmation processing on this conversation. The programs may issue calls and will recognize returned parameters relating to confirmation.

return_code
Specifies the result of the call execution, which is returned to the local program. The *return_code* variable can have one of the following values:

- CM_OK
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is not in **Initialize** state.

- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *sync_level* is set to an undefined value.
  - The *sync_level* is set to CM_NONE and *send_type* is set to CM_SEND_AND_CONFIRM.
  - The *sync_level* is set to CM_NONE and *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_CONFIRM.
  - The *sync_level* is set to CM_NONE and *deallocate_type* is set to CM_DEALLOCATE_CONFIRM.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

If a *return_code* other than CM_OK is returned on the call, the *sync_level* conversation characteristic is unchanged.

Related Information

"Confirm (CMCFM)" on page 52 and "Confirmed (CMCFMD)" on page 54 provide further information on confirmation processing.

Set_TP_Name (CMSTPN)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

Set_TP_Name (CMSTPN) is used by a program to set the *TP_name* characteristic for a given conversation. Set_TP_Name overrides the current value, which was originally acquired from the side information using the *sym_dest_name*. See "Side Information" on page 11 for more information.

This call does not change the value of *TP_name* in the side information. Set_TP_Name only changes the *TP_name* characteristic for this conversation.

Note: A program cannot issue Set_TP_Name after an Allocate is issued. Only a program that initiates a conversation (using the Initialize_Conversation call) can issue this call.

Format

```
CALL CMSTPN(conversation_ID,
            TP_name,
            TP_name_length,
            return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier.

TP_name
Specifies the name of the remote program. A program with the appropriate privilege can specify the name of an SNA service transaction program.

TP_name_length
Specifies the length of *TP_name*. The length can be from 1 to 64 bytes.

return_code
Specifies the result of the call execution, which is returned to the local program. The *return_code* can have one of the following values:

- CM_OK
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is not in **Initialize** state.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *TP_name_length* specifies a value less than 1 or greater than 64.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

If a *return_code* other than CM_OK is returned on the call, the *TP_name* and *TP_name_length* conversation characteristics are unchanged.

Related Information

See "SNA Service Transaction Programs" on page 149 for more information on privilege and service transaction programs.

"Side Information" on page 11 and note 3 of Table 6 on page 129 provide further discussion of the *TP_name* conversation characteristic.

Test_Request_To_Send_Received (CMTRTS)

TSO/E	CMS	OS/400	OS/2	IMS	CICS
	X				

Test_Request_To_Send_Received (CMTRTS) is used by a program to determine whether a request-to-send notification has been received from the remote program for the specified conversation.

Format

```
CALL CMTRTS(conversation_ID,
            request_to_send_received,
            return_code)
```

Parameters

conversation_ID
Specifies the conversation identifier.

request_to_send_received
Specifies the variable in which is returned an indication of whether or not a request-to-send notification has been received. The *request_to_send_received* variable can have one of the following values:

- CM_REQ_TO_SEND_RECEIVED
  A request-to-send notification has been received from the remote program. The remote program has issued Request_To_Send, requesting the local program to enter **Receive** state, which will place the remote program in **Send** state.
- CM_REQ_TO_SEND_NOT_RECEIVED
  A request-to-send notification has not been received.

Note: If *return_code* is either CM_PROGRAM_PARAMETER_CHECK or CM_PROGRAM_STATE_CHECK, the *request_to_send_received* parameter will not be replaced with a valid value and should not be examined by the program.

return_code
Specifies the result of the call execution, which is returned to the local program. The *return_code* variable can have one of the following values:

- CM_OK
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is not in **Send**, **Receive**, or **Send-Pending** state.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. When the local LU receives the request-to-send notification, it retains the notification until the local program issues a call (such as Test_Request_To_Send_Received) with the *request_to_send_received* parameter. It will retain only one request-to-send notification at a time (per conversation). Additional notifications are discarded until the retained notification is indicated to the local program. Therefore, a local program may issue the Request_To_Send call more times than are indicated to the remote program.
2. After the retained notification is indicated to the local program via the *request_to_send_received* parameter, the local LU discards the notification.

Related Information

"Request_To_Send (CMRTS)" on page 81 provides further discussion of the request-to-send function.

Appendix A. Variables and Characteristics

For the variables and characteristics used throughout this book, this appendix provides the following items:

- A chart showing the values that variables and characteristics can take. The valid pseudonyms and corresponding integer values are provided for each variable or characteristic.
- The character sets used by CPI Communications, along with their corresponding hexadecimal encodings.
- The data definitions for types and lengths of all CPI-Communications characteristics and variables.

Pseudonyms and Integer Values

As explained in "Naming Conventions — Calls and Characteristics, Variables and Values" on page 19, the values for variables and conversation characteristics have been shown as pseudonyms rather than integer values. For example, instead of stating that the variable *return_code* is set to an integer value of 0, the book shows the *return_code* being set to a pseudonym value of CM_OK. Table 4 on page 124 provides a mapping from valid pseudonyms to integer values for each variable or characteristic.

This same method of pseudonyms-for-integer-values can be used in program code by making use of equate or define statements. CPI Communications provides sample pseudonym files for each of the SAA languages — see "Programming Language Considerations" on page 177 for the names of the pseudonym files and Appendix F, "Sample Programs" on page 181 for an example of how one is used by a COBOL program.

Note: Because the *return_code* variable is used for all CPI-Communications calls, Appendix B, "Return Codes" provides a more detailed description of its values, in addition to the list of values provided here.

Table 4 (Page 1 of 2). Variables/Characteristics and Their Possible Values

Variable or Characteristic Name	Pseudonym Values	Integer Values
conversation_type	CM_BASIC_CONVERSATION	0
	CM_MAPPED_CONVERSATION	1
data_received	CM_NO_DATA_RECEIVED	0
	CM_DATA_RECEIVED	1
	CM_COMPLETE_DATA_RECEIVED	2
	CM_INCOMPLETE_DATA_RECEIVED	3
deallocate_type	CM_DEALLOCATE_SYNC_LEVEL	0
	CM_DEALLOCATE_FLUSH	1
	CM_DEALLOCATE_CONFIRM	2
	CM_DEALLOCATE_ABEND	3
error_direction	CM_RECEIVE_ERROR	0
	CM_SEND_ERROR	1
fill	CM_FILL_LL	0
	CM_FILL_BUFFER	1
prepare_to_receive_type	CM_PREP_TO_RECEIVE_SYNC_LEVEL	0
	CM_PREP_TO_RECEIVE_FLUSH	1
	CM_PREP_TO_RECEIVE_CONFIRM	2
receive_type	CM_RECEIVE_AND_WAIT	0
	CM_RECEIVE_IMMEDIATE	1
request_to_send_received	CM_REQ_TO_SEND_NOT_RECEIVED	0
	CM_REQ_TO_SEND_RECEIVED	1

Table 4 (Page 2 of 2). Variables/Characteristics and Their Possible Values

Variable or Characteristic Name	Pseudonym Values	Integer Values
return_code	CM_OK	0
	CM_ALLOCATE_FAILURE_NO_RETRY	1
	CM_ALLOCATE_FAILURE_RETRY	2
	CM_CONVERSATION_TYPE_MISMATCH	3
	CM_SECURITY_NOT_VALID	6
	CM_SYNC_LVL_NOT_SUPPORTED_PGM	8
	CM_TPN_NOT_RECOGNIZED	9
	CM_TP_NOT_AVAILABLE_NO_RETRY	10
	CM_TP_NOT_AVAILABLE_RETRY	11
	CM_DEALLOCATED_ABEND	17
	CM_DEALLOCATED_NORMAL	18
	CM_PARAMETER_ERROR	19
	CM_PRODUCT_SPECIFIC_ERROR	20
	CM_PROGRAM_ERROR_NO_TRUNC	21
	CM_PROGRAM_ERROR_PURGING	22
	CM_PROGRAM_ERROR_TRUNC	23
	CM_PROGRAM_PARAMETER_CHECK	24
	CM_PROGRAM_STATE_CHECK	25
	CM_RESOURCE_FAILURE_NO_RETRY	26
	CM_RESOURCE_FAILURE_RETRY	27
	CM_UNSUCCESSFUL	28
return_control	CM_WHEN_SESSION_ALLOCATED	0
	CM_IMMEDIATE	1
send_type	CM_BUFFER_DATA	0
	CM_SEND_AND_FLUSH	1
	CM_SEND_AND_CONFIRM	2
	CM_SEND_AND_PREP_TO_RECEIVE	3
	CM_SEND_AND_DEALLOCATE	4
status_received	CM_NO_STATUS_RECEIVED	0
	CM_SEND_RECEIVED	1
	CM_CONFIRM_RECEIVED	2
	CM_CONFIRM_SEND_RECEIVED	3
	CM_CONFIRM_DEALLOC_RECEIVED	4
sync_level	CM_NONE	0
	CM_CONFIRM	1

Character Sets

CPI Communications makes use of character strings composed of characters from one of the following character sets:

- Character set 01134, which is composed of the uppercase letters A through Z, numerals 0-9.
- Character set 00640, which is composed of the uppercase and lowercase letters A through Z, numerals 0-9, and 20 special characters.

These character sets, along with hexadecimal and graphic representations, are provided in Table 5. See *SNA Formats* for more information on character sets.

Table 5 (Page 1 of 2). Character Sets 01134 and 00640

Hex Code	Graphic	Description	Character Set	
			01134	00640
40		Blank		X
4B	.	Period		X
4C	<	Less than sign		X
4D	(	Left parenthesis		X
4E	+	Plus sign		X
50	&	Ampersand		X
5C	*	Asterisk		X
5D	)	Right parenthesis		X
5E	;	Semi-colon		X
60	-	Dash		X
61	/	Slash		X
6B	,	Comma		X
6C	%	Percent sign		X
6D	_	Underscore		X
6E	>	Greater than sign		X
6F	?	Question mark		X
7A	:	Colon		X
7D	'	Single quote		X
7E	=	Equal sign		X
7F	"	Double quote		X
81	a	Lowercase a		X
82	b	Lowercase b		X
83	c	Lowercase c		X
84	d	Lowercase d		X
85	e	Lowercase e		X
86	f	Lowercase f		X
87	g	Lowercase g		X
88	h	Lowercase h		X
89	i	Lowercase i		X
91	j	Lowercase j		X
92	k	Lowercase k		X
93	l	Lowercase l		X
94	m	Lowercase m		X
95	n	Lowercase n		X
96	o	Lowercase o		X
97	p	Lowercase p		X
98	q	Lowercase q		X
99	r	Lowercase r		X

Table 5 (Page 2 of 2). Character Sets 01134 and 00640

Hex Code	Graphic	Description	Character Set	
			01134	00640
A2	s	Lowercase s		X
A3	t	Lowercase t		X
A4	u	Lowercase u		X
A5	v	Lowercase v		X
A6	w	Lowercase w		X
A7	x	Lowercase x		X
A8	y	Lowercase y		X
A9	z	Lowercase z		X
C1	A	Uppercase A	X	X
C2	B	Uppercase B	X	X
C3	C	Uppercase C	X	X
C4	D	Uppercase D	X	X
C5	E	Uppercase E	X	X
C6	F	Uppercase F	X	X
C7	G	Uppercase G	X	X
C8	H	Uppercase H	X	X
C9	I	Uppercase I	X	X
D1	J	Uppercase J	X	X
D2	K	Uppercase K	X	X
D3	L	Uppercase L	X	X
D4	M	Uppercase M	X	X
D5	N	Uppercase N	X	X
D6	O	Uppercase O	X	X
D7	P	Uppercase P	X	X
D8	Q	Uppercase Q	X	X
D9	R	Uppercase R	X	X
E2	S	Uppercase S	X	X
E3	T	Uppercase T	X	X
E4	U	Uppercase U	X	X
E5	V	Uppercase V	X	X
E6	W	Uppercase W	X	X
E7	X	Uppercase X	X	X
E8	Y	Uppercase Y	X	X
E9	Z	Uppercase Z	X	X
F0	0	Zero	X	X
F1	1	One	X	X
F2	2	Two	X	X
F3	3	Three	X	X
F4	4	Four	X	X
F5	5	Five	X	X
F6	6	Six	X	X
F7	7	Seven	X	X
F8	8	Eight	X	X
F9	9	Nine	X	X

Variable Types

CPI Communications makes use of two variable types, integer and character string. Table 6 on page 129 defines the type and length of variables used in this document. Specifics on the types are provided below.

Integers

The integers are signed non-negative integers. Their length is provided in bits.

Character Strings

Character-strings are composed of characters taken from one of the character sets discussed in "Character Sets" on page 126, or, in the case of *buffer*, are bytes with no restrictions (that is, a string composed of characters from X'00' to X'FF').

Note: The name "character string" as used in this manual should not be confused with "character string" as used in the C programming language. No further restrictions beyond those described above are intended.

The character-string length represents the number of characters a character string can contain. Two character-string lengths are defined for each variable of type character-string:

- **Minimum specification length:** the minimum number of characters that a program can use to specify the character string. For some character strings, the minimum specification length is zero. A zero-length character string on a call means the character string is omitted, regardless of the length of the variable that contains the character string (see the notes for Table 6 on page 129).
- **Maximum specification length:** the maximum number of characters that a transaction program can use to specify a character string. All products can send or receive the maximum-specification length for the character string.

For example, the character-string length for *log_data* is listed as 0-512 bytes where 0 is the minimum specification length and 512 is the maximum specification length.

If the variable to which a character string is assigned is longer than the character string, the character string is left-justified within the variable and the variable is filled out to the right with space (X'40') characters. Space characters, if present, are not part of the character string.

If the character string is formed from the concatenation of two or more individual character strings, the concatenated character string as a whole is left-justified within the variable and the variable is filled out to the right with space (X'40') characters. Space characters, if present, are not part of the concatenated character string.

Table 6. Variable Types and Lengths

Variable	Variable Type	Character Set	Length
buffer	Character string	no restriction	0-32767 bytes
conversation_ID	Character string	no restriction	8 bytes
conversation_type	Integer	N/A	32 bits
send_length	Integer	N/A	32 bits
data_received	Integer	N/A	32 bits
deallocate_type	Integer	N/A	32 bits
error_direction	Integer	N/A	32 bits
fill	Integer	N/A	32 bits
log_data	Character string	00640	0-512 bytes
log_data_length	Integer	N/A	32 bits
mode_name [1]	Character string	01134	0-8 bytes
mode_name_length	Integer	N/A	32 bits
partner_LU_name [1, 2]	Character string	01134	1-17 bytes
partner_LU_name_length	Integer	N/A	32 bits
prepare_to_receive_type	Integer	N/A	32 bits
receive_type	Integer	N/A	32 bits
received_length	Integer	N/A	32 bits
requested_length	Integer	N/A	32 bits
request_to_send_received	Integer	N/A	32 bits
return_code	Integer	N/A	32 bits
return_control	Integer	N/A	32 bits
send_type	Integer	N/A	32 bits
status_received	Integer	N/A	32 bits
sync_level	Integer	N/A	32 bits
sym_dest_name	Character string	01134	8 bytes
TP_name [3]	Character string	00640	1-64 bytes
TP_name_length	Integer	N/A	32 bits

Notes:

1. Because the *mode_name* and *partner_LU_name* characteristics are output parameters on their respective Extract calls, the variables used to contain the output character strings should be defined with a length equal to the maximum specification length.
2. The *partner_LU_name* can be of two varieties:
   - A character string composed solely of characters drawn from character set 01134
   - A character string consisting of two character strings composed of characters drawn from character set 01134. The two character strings are concatenated together by a period (the period is not part of character set 01134). The left-hand character string represents the network ID, and the right-hand character string represents the network LU name. The period is not part of the network ID or the network LU name. Neither network ID nor network LU name may be longer than eight bytes in length.

   The use of the period differentiates between which variety of *partner_LU_name* is being used.

   On VM, a blank is used as a delimiter instead of a period.
3. When communicating with non-CPI-Communications programs, the *TP_name* can use characters other than those in character set 00640. See Appendix D, "CPI Communications and LU 6.2" on page 147 and "SNA Service Transaction Programs" on page 149 for details.

Appendix B. Return Codes

All calls have a parameter called *return_code* that is used to pass a return code back to the program at the completion of a call. The return code indicates the result of call execution and includes any state changes to the specified conversation.

Some of the return codes indicate the results of the local processing of a call. These return codes are returned on the call that invoked the local processing. Other return codes indicate results of processing invoked at the remote end of the conversation. Depending on the call, these return codes can be returned on the call that invoked the remote processing or on a subsequent call. Still other return codes report events that originate at the remote end of the conversation. In all cases, only one code is returned at a time.

Some of the return codes associated with the allocation of a conversation have the suffix RETRY or NO_RETRY in their name. RETRY means that the condition indicated by the return code may not be permanent, and the program can try to allocate the conversation again. Whether or not the retry attempt succeeds depends on the duration of the condition. In general, the program should limit the number of times it attempts to retry without success. NO_RETRY means that the condition is probably permanent. In general, a program should not attempt to allocate the conversation again until the condition is corrected.

The return codes shown below are listed alphabetically, and each description includes the following:

- The meaning of the return code
- The origin of the condition indicated by the return code
- When the return code is reported to the program
- The state of the conversation when control is returned to the program.

Notes:

1. The individual call descriptions in Chapter 4, "Reference Section" list the return code values that are valid for each call.
2. The integer values that correspond to the pseudonyms listed below are provided in Table 4 on page 124 of Appendix A, "Variables and Characteristics."

The valid *return_code* values are described below:

CM_ALLOCATE_FAILURE_NO_RETRY
: The conversation cannot be allocated on a session because of a condition that is not temporary. When this *return_code* value is returned to the program, the conversation is in **Reset** state. For example, the session to be used for the conversation cannot be activated because the current (LU,mode) session limit for the specified (LU-name,mode-name) pair is 0, or because of a system definition error or a session-activation protocol error. This return code is also returned when the session is deactivated because of a session protocol error before the conversation can be allocated. The program should not retry the allocation request until the condition is corrected. This return code is returned on the Allocate call.

CM_ALLOCATE_FAILURE_RETRY
: The conversation cannot be allocated on a session because of a condition that may be temporary. When this *return_code* value is returned to the program, the conversation is in **Reset** state. For example, the session to be used for the conversation cannot be activated because of a temporary lack of resources at the local LU or remote LU. This return code is also returned if the session is deactivated because of session outage before the conversation can be allocated. The program can retry the allocation request. This return code is returned on the Allocate call.

CM_CONVERSATION_TYPE_MISMATCH
: The remote LU rejected the allocation request because the local program issued an Allocate call with *conversation_type* set to either CM_MAPPED_CONVERSATION or CM_BASIC_CONVERSATION, and the remote program does not support the respective mapped or basic conversation protocol boundary. This return code is returned on a call subsequent to the Allocate. When this *return_code* value is returned to the program, the conversation is in **Reset** state.

CM_DEALLOCATED_ABEND
: The remote program issued a Deallocate call with *deallocate_type* (CM_DEALLOCATE_ABEND) in effect, or the remote LU has done so because of a remote program abnormal-ending condition. If the conversation for the remote program was in **Receive** state when the call was issued, information sent by the local program and not yet received by the remote program is purged. This return code is also reported to the local program on a call the program issues in **Send** or **Receive** state. The conversation is in **Reset** state.

CM_DEALLOCATED_NORMAL
: The remote program issued a Deallocate call on a basic or mapped conversation with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL or CM_DEALLOCATE_FLUSH. If *deallocate_type* is CM_DEALLOCATE_SYNC_LEVEL, the synchronization level is CM_NONE. This return code is reported to the local program on a call the program issues in **Receive** state. The conversation is in **Reset** state.

CM_OK
: The call issued by the local program executed successfully (that is, the function defined for the call, up to the point at which control is returned to the program, was performed as specified). The state of the conversation is as defined for the call.

CM_PARAMETER_ERROR
: The local program issued a call specifying a parameter containing an invalid argument. ("Parameters" include not only the parameters described as part of the call syntax, but also characteristics associated with the *conversation_ID*.) The source of the argument is considered to be outside the program definition, such as an LU name supplied by a system administrator in the side information and referenced by the Initialize_Conversation call.

  The CM_PARAMETER_ERROR return code is returned on the call specifying the invalid argument. The state of the conversation remains unchanged.

  Note: Contrast this definition with the definition of the CM_PROGRAM_PARAMETER_CHECK return code.

CM_PRODUCT_SPECIFIC_ERROR
A product-specific error has been detected and a description of the error has been entered into the product's system error log. See product documentation for an indication of conditions and state changes caused by this return code.

CM_PROGRAM_ERROR_NO_TRUNC
One of the following occurred:

- The remote program issued a Send_Error call on a mapped conversation and the conversation for the remote program was in **Send** state. No truncation occurs at the mapped conversation protocol boundary. This return code is reported to the local program on a Receive call the program issues before receiving any data records or after receiving one or more data records.
- The remote program issued a Send_Error call on a basic conversation, the conversation for the remote program was in **Send** state, and the call did not truncate a logical record. No truncation occurs at the basic conversation protocol boundary when a program issues Send_Error before sending any logical records or after sending a complete logical record. This return code is reported to the local program on a Receive call the program issues before receiving any logical records or after receiving one or more complete logical records.
- The remote program issued a Send_Error call on a mapped or basic conversation and the conversation for the remote program was in **Send-Pending** state. No truncation of data has occurred. This return code indicates that the remote program has issued Set_Error_Direction to set the *error_direction* characteristic to CM_SEND_ERROR. The return code is reported to the local program on a Receive call the program issues before receiving any data records or after receiving one or more data records.

The conversation remains in **Receive** state.

CM_PROGRAM_ERROR_PURGING
One of the following occurred:

- The remote program issued a Send_Error call on a basic or mapped conversation and the conversation for the remote program was in **Receive** or **Confirm** state. The call may have caused information to be purged. Purging occurs when a program issues Send_Error in **Receive** state before receiving all the information sent by its partner program (all of the information sent before reporting the CM_PROGRAM_ERROR_PURGING return code to the partner program). The purging can occur at the local LU, remote LU, or both. No purging occurs when a program issues the call in **Confirm** state, or in **Receive** state after receiving all the information sent by its partner program.
- The remote program issued a Send_Error call on a mapped or basic conversation and the conversation for the remote program was in **Send-Pending** state. No purging of data has occurred. This return code indicates that the remote program had an *error_direction* characteristic set to CM_RECEIVE_ERROR when the Send_Error call was made.

This return code is normally reported to the local program on a call the program issues after sending some information to the remote program. However, the return code can be reported on a call the program issues before sending any information, depending on the call and when it is issued. The conversation remains in **Receive** state.

CM_PROGRAM_ERROR_TRUNC
: The remote program issued a Send_Error call on a basic conversation, the conversation for the remote program was in **Send** state, and the call truncated a logical record. Truncation occurs at the basic conversation protocol boundary when a program begins sending a logical record and then issues Send_Error before sending the complete logical record. This return code is reported to the local program on a Receive call the program issues after receiving the truncated logical record. The conversation remains in **Receive** state.

CM_PROGRAM_PARAMETER_CHECK
: The local program issued a call in which a programming error has been found in one or more parameters. ("Parameters" include not only the parameters described as part of the call syntax, but also characteristics associated with the *conversation_ID*.) The source of the error is considered to be inside the program definition (under the control of the local program). The program should not examine any other returned variables associated with the call as nothing is placed in the variables. The state of the conversation remains unchanged.

CM_PROGRAM_STATE_CHECK
: The local program issued a call in a state that was not valid for that call. The program should not examine any other returned variables associated with the call as nothing is placed in the variables. The state of the conversation remains unchanged.

CM_RESOURCE_FAILURE_NO_RETRY
: A failure occurred that caused the conversation to be prematurely terminated. For example, the session being used for the conversation was deactivated because of a session protocol error, or the conversation was deallocated because of a protocol error between the mapped conversation components of the LUs. The condition is not temporary, and the program should not retry the transaction until the condition is corrected. This return code can be reported to the local program on a call it issues in any state other than **Reset**. The conversation is in **Reset** state.

CM_RESOURCE_FAILURE_RETRY
: A failure occurred that caused the conversation to be prematurely terminated. For example, the session being used for the conversation was deactivated because of a session outage such as a line failure, a modem failure, or a crypto engine failure. The condition may be temporary, and the program can retry the transaction. This return code can be reported to the local program on a call it issues in any state other than **Reset**. The conversation is in **Reset** state.

CM_SECURITY_NOT_VALID
: The remote LU rejected the allocation request because the access security information (provided by the local system) is invalid. This return code is returned on a call made after the Allocate. When this *return_code* value is returned to the program, the conversation is in **Reset** state.

CM_SYNC_LVL_NOT_SUPPORTED_PGM
: The remote LU rejected the allocation request because the local program specified a synchronization level (with the *sync_level* parameter) that the remote program does not support. This return code is returned on a call made after the Allocate. When this *return_code* value is returned to the program, the conversation is in **Reset** state.

CM_TPN_NOT_RECOGNIZED
: The remote LU rejected the allocation request because the local program specified a remote program name that the remote LU does not recognize. This return code is returned on a call made after the Allocate. When this *return_code* value is returned to the program, the conversation is in **Reset** state.

CM_TP_NOT_AVAILABLE_NO_RETRY
: The remote LU rejected the allocation request because the local program specified a remote program that the remote LU recognizes but cannot start. The condition is not temporary, and the program should not retry the allocation request. This return code is returned on a call made after the Allocate. When this *return_code* value is returned to the program, the conversation is in **Reset** state.

CM_TP_NOT_AVAILABLE_RETRY
: The remote LU rejected the allocation request because the local program specified a remote program that the remote LU recognizes but currently cannot start. The condition may be temporary, and the program can retry the allocation request. This return code is returned on a call made after the Allocate. When this *return_code* value is returned to the program, the conversation is in **Reset** state.

CM_UNSUCCESSFUL
: The call issued by the local program did not execute successfully. This return code is returned on the unsuccessful call. The state of the conversation remains unchanged.

Appendix C. State Table

The CPI-Communications state table shows when and where different CPI-Communications calls can be issued. For example, a program must issue an Initialize_Conversation call before issuing an Allocate call, and it cannot issue a Send_Data call before the conversation is allocated.

As described in "Program Flow — States and Transitions" on page 17, CPI Communications uses the concepts of states and state transitions to simplify explanations of the restrictions that are placed on the calls. A number of states are defined for CPI Communications and, for any given call, a number of transitions are allowed. Table 7 on page 142 shows the state table, which describes the state transitions that are allowed for the CPI-Communications calls.

Explanation of State-Table Abbreviations

Abbreviations are used in the state table to indicate the different permutations of calls and characteristics. There are three categories of abbreviations:

- **Conversation characteristic** abbreviations are enclosed by parenthesis — "(. . .)"
- **return_code** abbreviations are enclosed by brackets — "[. . .]"
- **data_received and status_received** abbreviations are enclosed by braces and separated by a comma — "{ . . . , . . . }" — where the abbreviation before the comma represents the *data_received* value and the abbreviation after the comma represents the value of *status_received*.

The next sections show the abbreviations used in each category.

Conversation Characteristics ( )

The following abbreviations are used for conversation characteristics:

Abbreviation	Meaning
A	*deallocate_type* is set to CM_DEALLOCATE_ABEND
B	*send_type* is set to CM_BUFFER_DATA
C	For a Deallocate call, C means one of the following: • *deallocate_type* is set to CM_DEALLOCATE_CONFIRM • *deallocate_type* is set to CM_DEALLOCATE_SYNC_LVL and *sync_level* is set to CM_CONFIRM. For a Prepare_To_Receive, C means one of the following: • *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_CONFIRM • *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* is set to CM_CONFIRM For a Send_Data call, C means the following: • *send_type* is set to CM_SEND_AND_CONFIRM
D	*send_type* is set to CM_SEND_AND_DEALLOCATE.
F	For a Deallocate call, F means one of the following: • *deallocate_type* is set to CM_DEALLOCATE_FLUSH • *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is set to CM_NONE. For a Prepare_To_Receive call, F means one of the following: • *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_FLUSH • *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* is set to CM_NONE. For a Send_Data call, F means the following: • *send_type* is set to CM_SEND_AND_FLUSH
I	*receive_type* is set to CM_RECEIVE_IMMEDIATE
P	*send_type* is set to CM_SEND_AND_PREP_TO_RECEIVE
W	*receive_type* is set to CM_RECEIVE_AND_WAIT.

Return Code Values [ ]

The following abbreviations are used for return codes:

Abbreviation	Meaning
ae	For an Allocate call, ae means one of the following: • CM_ALLOCATE_FAILURE_NO_RETRY • CM_ALLOCATE_FAILURE_RETRY For any other call, ae means one of the following: • CM_CONVERSATION_TYPE_MISMATCH • CM_SECURITY_NOT_VALID • CM_SYNC_LEVEL_NOT_SUPPORTED_PGM • CM_TPN_NOT_RECOGNIZED • CM_TP_NOT_AVAILABLE_NO_RETRY • CM_TP_NOT_AVAILABLE_RETRY
da	CM_DEALLOCATED_ABEND
dn	CM_DEALLOCATED_NORMAL
en	CM_PROGRAM_ERROR_NO_TRUNC
ep	CM_PROGRAM_ERROR_PURGING
et	CM_PROGRAM_ERROR_TRUNC
ok	CM_OK
pe	CM_PARAMETER_ERROR
pc	CM_PROGRAM_PARAMETER_CHECK
ps	CM_PRODUCT_SPECIFIC_ERROR
rf	CM_RESOURCE_FAILURE_NO_RETRY or CM_RESOURCE_FAILURE_RETRY
un	CM_UNSUCCESSFUL

data_received and status_received { , }

The following abbreviations are used for the *data_received* values:

Abbreviation	Meaning
dr	Means one of the following: • CM_DATA_RECEIVED • CM_COMPLETE_DATA_RECEIVED • CM_INCOMPLETE_DATA_RECEIVED
nd	CM_NO_DATA_RECEIVED
*	Means one of the following: • CM_DATA_RECEIVED • CM_COMPLETE_DATA_RECEIVED • CM_NO_DATA_RECEIVED

The following abbreviations are used for the *status_received* values:

Abbreviation	Meaning
cd	CM_CONFIRM_DEALLOC_RECEIVED
cs	CM_CONFIRM_SEND_RECEIVED
co	CM_CONFIRM_RECEIVED
no	CM_NO_STATUS_RECEIVED
se	CM_SEND_RECEIVED

Table Symbols

The following symbols are used in the state table to indicate the condition that results when a call is issued from a certain state:

Symbol	Meaning
/	"cannot occur" situation
–	Remain in current state
1-8	Number of next state
>	State error. A *return_code* of CM_PROGRAM_STATE_CHECK is returned. For calls illegally issued in **Reset** state, this condition is indicated to the program with a return code of CM_PROGRAM_PARAMETER_CHECK. This is because the program is in **Reset** state and the *conversation_ID* for the conversation is undefined.

How to Use the State Table

The various calls and combinations of parameters, also referred to as *inputs*, are shown along the left side of the table. These inputs correspond to the rows of the table. The possible states are shown across the top of the table. The states correspond to the columns of the matrix. The intersection of input (row) and state (column) represents what state transition, if any, will occur for the CPI-Communications call that is issued in that particular state.

For example, look at `Initialize_Conversation[ok]`. The `[ok]` indicates that a return code of CM_OK was received on the call. By examining the row, it is seen that the call can only be issued in **Reset** state (state 1).

When issued in state 1, the 2 in the column for **Reset** indicates that the program progresses to state 2, **Initialize**. A scan down this column shows that the Allocate call can be made from here. The two variations in the Allocate row that entail a change of state are for return codes of:

- "ok," indicating a *return_code* of CM_OK, which allows the program to progress to state 3.
- "ae," indicating a *return_code* of CM_ALLOCATE_FAILURE_NO_RETRY or CM_ALLOCATE_FAILURE_RETRY, which puts the program back into reset state (state 1).

Table 7 (Page 1 of 4). States and Transitions for CPI-Communications Calls

Inputs	Reset 1	Initialize 2	Send 3	Receive 4	Send Pending 5	Confirm 6	Confirm Send 7	Confirm Deallocate 8
Initialize_Conversation[ok]	2	/	/	/	/	/	/	/
Initialize_Conversation[pc]	–	/	/	/	/	/	/	/
Initialize_Conversation[ps]	–	/	/	/	/	/	/	/
Accept_Conversation[ok]	4	/	/	/	/	/	/	/
Accept_Conversation[pc]	–	/	/	/	/	/	/	/
Accept_Conversation[ps]	–	/	/	/	/	/	/	/
Allocate[ok]	>	3	>	>	>	>	>	>
Allocate[ae]	>	1	>	>	>	>	>	>
Allocate[pc]	>	–	>	>	>	>	>	>
Allocate[pe]	>	–	>	>	>	>	>	>
Allocate[ps]	>	–	>	>	>	>	>	>
Allocate[un]	>	–	>	>	>	>	>	>
Confirm[ok]	>	>	–	>	3	>	>	>
Confirm[ae]	>	>	1	>	1	>	>	>
Confirm[da]	>	>	1	>	1	>	>	>
Confirm[ep]	>	>	4	>	4	>	>	>
Confirm[rf]	>	>	1	>	1	>	>	>
Confirm[pc]	>	>	–	>	–	>	>	>
Confirm[ps]	>	>	–	>	–	>	>	>
Confirmed[ok]	>	>	>	>	>	4	3	1
Confirmed[pc]	>	>	>	>	>	–	–	–
Confirmed[ps]	>	>	>	>	>	–	–	–
Extract_Conversation_Type[ok]	>	–	–	–	–	–	–	–
Extract_Mode_Name[ok]	>	–	–	–	–	–	–	–
Extract_Partner_LU_Name[ok]	>	–	–	–	–	–	–	–
Extract_Sync_Level[ok]	>	–	–	–	–	–	–	–
Deallocate(F) [ok]	>	>	1	>	1	>	>	>
Deallocate(F) [pc]	>	>	–	>	–	>	>	>
Deallocate(F) [ps]	>	>	–	>	–	>	>	>
Deallocate(C) [ok]	>	>	1	>	1	>	>	>
Deallocate(C) [ae]	>	>	1	>	1	>	>	>
Deallocate(C) [da]	>	>	1	>	1	>	>	>
Deallocate(C) [ep]	>	>	4	>	4	>	>	>
Deallocate(C) [rf]	>	>	1	>	1	>	>	>
Deallocate(C) [pc]	>	>	–	>	–	>	>	>
Deallocate(C) [ps]	>	>	–	>	–	>	>	>

Table 7 (Page 2 of 4). States and Transitions for CPI-Communications Calls

Inputs	Reset 1	Initialize 2	Send 3	Receive 4	Send Pending 5	Confirm 6	Confirm Send 7	Confirm Deallocate 8
Deallocate(A) [ok]	>	1	1	1	1	1	1	1
Deallocate(A) [pc]	>	–	–	–	–	–	–	–
Deallocate(A) [ps]	>	–	–	–	–	–	–	–
Flush[ok]	>	>	–	>	3	>	>	>
Flush[pc]	>	>	–	>	–	>	>	>
Flush[ps]	>	>	–	>	–	>	>	>
Prepare_To_Receive(F) [ok]	>	>	4	>	4	>	>	>
Prepare_To_Receive(F) [pc]	>	>	–	>	–	>	>	>
Prepare_To_Receive(F) [ps]	>	>	–	>	–	>	>	>
Prepare_To_Receive(C) [ok]	>	>	4	>	4	>	>	>
Prepare_To_Receive(C) [ae]	>	>	1	>	1	>	>	>
Prepare_To_Receive(C) [da]	>	>	1	>	1	>	>	>
Prepare_To_Receive(C) [ep]	>	>	4	>	4	>	>	>
Prepare_To_Receive(C) [rf]	>	>	1	>	1	>	>	>
Prepare_To_Receive(C) [pc]	>	>	–	>	–	>	>	>
Prepare_To_Receive(C) [ps]	>	>	–	>	–	>	>	>
Receive(W) [ok] {dr,no}	>	>	4	–	4	>	>	>
Receive(W) [ok] {nd,se}	>	>	3	3	/	>	>	>
Receive(W) [ok] {dr,se}	>	>	5	5	/	>	>	>
Receive(W) [ok] {*,co}	>	>	6	6	/	>	>	>
Receive(W) [ok] {*,cs}	>	>	7	7	/	>	>	>
Receive(W) [ok] {*,cd}	>	>	8	8	/	>	>	>
Receive(W) [ae]	>	>	1	1	/	>	>	>
Receive(W) [da]	>	>	1	1	1	>	>	>
Receive(W) [dn]	>	>	1	1	1	>	>	>
Receive(W) [en]	>	>	4	–	4	>	>	>
Receive(W) [ep]	>	>	4	–	4	>	>	>
Receive(W) [et]	>	>	/	–	/	>	>	>
Receive(W) [rf]	>	>	1	1	1	>	>	>
Receive(W) [pc]	>	>	–	–	–	>	>	>
Receive(W) [ps]	>	>	–	–	–	>	>	>
Receive(I) [ok] {dr,no}	>	>	>	–	>	>	>	>
Receive(I) [ok] {nd,se}	>	>	>	3	>	>	>	>
Receive(I) [ok] {dr,se}	>	>	>	5	>	>	>	>
Receive(I) [ok] {*,co}	>	>	>	6	>	>	>	>
Receive(I) [ok] {*,cs}	>	>	>	7	>	>	>	>
Receive(I) [ok] {*,cd}	>	>	>	8	>	>	>	>

Table 7 (Page 3 of 4). States and Transitions for CPI-Communications Calls

Inputs	Reset 1	Initialize 2	Send 3	Receive 4	Send Pending 5	Confirm 6	Confirm Send 7	Confirm Deallocate 8
Receive(I) [ae]	>	>	>	1	>	>	>	>
Receive(I) [da]	>	>	>	1	>	>	>	>
Receive(I) [dn]	>	>	>	1	>	>	>	>
Receive(I) [en]	>	>	>	–	>	>	>	>
Receive(I) [ep]	>	>	>	–	>	>	>	>
Receive(I) [et]	>	>	>	–	>	>	>	>
Receive(I) [rf]	>	>	>	1	>	>	>	>
Receive(I) [pc]	>	>	>	–	>	>	>	>
Receive(I) [ps]	>	>	>	–	>	>	>	>
Receive(I) [un]	>	>	>	–	>	>	>	>
Request_To_Send[ok]	>	>	–	–	–	–	–	–
Request_To_Send[pc]	>	>	–	–	–	–	–	–
Request_To_Send[ps]	>	>	–	–	–	–	–	–
Send_Data(B) [ok]	>	>	–	>	3	>	>	>
Send_Data(F) [ok]	>	>	–	>	3	>	>	>
Send_Data(C) [ok]	>	>	–	>	3	>	>	>
Send_Data(P) [ok]	>	>	4	>	4	>	>	>
Send_Data(D) [ok]	>	>	1	>	1	>	>	>
Send_Data[ae]	>	>	1	>	1	>	>	>
Send_Data[da]	>	>	1	>	1	>	>	>
Send_Data[ep]	>	>	4	>	4	>	>	>
Send_Data[pc]	>	>	–	>	–	>	>	>
Send_Data[rf]	>	>	1	>	1	>	>	>
Send_Data[ps]	>	>	–	>	–	>	>	>
Send_Error[ok]	>	>	–	3	3	3	3	3
Send_Error[ae]	>	>	1	/	/	/	/	/
Send_Error[da]	>	>	1	/	/	/	/	/
Send_Error[dn]	>	>	/	1	/	/	/	/
Send_Error[ep]	>	>	4	/	/	/	/	/
Send_Error[pc]	>	>	–	–	–	–	–	–
Send_Error[rf]	>	>	1	1	1	1	1	1
Send_Error[ps]	>	>	–	–	–	–	–	–
Set_Conversation_Type[ok]	>	–	>	>	>	>	>	>
Set_Deallocate_Type[ok]	>	–	–	–	–	–	–	–
Set_Error_Direction[ok]	>	–	–	–	–	–	–	–
Set_Fill[ok]	>	–	–	–	–	–	–	–
Set_Log_Data[ok]	>	–	–	–	–	–	–	–
Set_Mode_Name[ok]	>	–	>	>	>	>	>	>

Table 7 (Page 4 of 4). States and Transitions for CPI-Communications Calls

Inputs	Reset 1	Initialize 2	Send 3	Receive 4	Send Pending 5	Confirm 6	Confirm Send 7	Confirm Deallocate 8
Set_Partner_LU_Name[ok]	>	–	>	>	>	>	>	>
Set_Prepare_To_Receive_Type[ok]	>	–	–	–	–	–	–	–
Set_Receive_Type[ok]	>	–	–	–	–	–	–	–
Set_Return_Control[ok]	>	–	>	>	>	>	>	>
Set_Send_Type[ok]	>	–	–	–	–	–	–	–
Set_Sync_Level[ok]	>	–	>	>	>	>	>	>
Set_TP_Name[ok]	>	–	>	>	>	>	>	>
Test[ok]	>	>	–	–	–	>	>	>
Test[pc]	>	>	–	–	–	>	>	>
Test[ps]	>	>	–	–	–	>	>	>

Appendix D. CPI Communications and LU 6.2

This appendix is intended for programmers who are familiar with the LU 6.2 application programming interface. (LU 6.2 is also known as Advanced Program-to-Program Communications or APPC.) It describes the functional relationship between the APPC "verbs" and the CPI-Communications calls described in this manual.

The CPI-Communications calls have been built on top of the LU 6.2 verbs described in *SNA Transaction Programmer's Reference Manual for LU Type 6.2*. Table 8 beginning on page 150 shows the relationship between APPC verbs and CPI-Communications calls. Use this table to determine how the function of a particular LU 6.2 verb is provided through CPI Communications.

Although much of the LU 6.2 function has been included in CPI Communications, some of the function has not. Likewise, CPI Communications contains new features that are not found in LU 6.2.

Note: These "new" features are differences in syntax. The semantics of LU 6.2 function have not been changed or extended.

CPI Communications contains the following new features:

- A conversation state of **Send-Pending** (discussed in more detail in "Send-Pending State and the error_direction Characteristic" on page 148).
- The Accept_Conversation call for use by a remote program to explicitly establish a conversation, the conversation identifier, and the conversation's characteristics.
- The *error_direction* conversation characteristic (discussed in more detail in "Send-Pending State and the error_direction Characteristic" on page 148).
- A *send_type* conversation characteristic for use in combining functions (this function was available with LU 6.2 verbs, but the verbs had to be issued separately).
- The capability to return both data and conversation status on the same Receive call.

CPI Communications does not support the following functions that are available with the LU 6.2 interface:

- PIP data
- LOCKS=LONG
- MAP_NAME
- FMH_DATA.
- Syncpoint

Not listed above are the LU 6.2 security parameters. A set of LU 6.2 security parameters is established for the conversation (currently using a default value of SECURITY=SAME), but CPI Communications does not provide a method for the program to modify or examine the security parameters.

Finally, to increase portability between systems, the character sets used to specify the partner *TP_name*, *partner_LU_name*, and *log_data* have been modified slightly from the character sets allowed by LU 6.2. To answer specific questions of compatibility, check the character sets described in Appendix A, "Variables and Characteristics."

Send-Pending State and the error_direction Characteristic

The **Send-Pending** state and *error_direction* characteristic are part of CPI Communications because, using CPI Communications, a program may receive data and a change-of-direction indication at the same time. This "double function" creates a possibly ambiguous error condition since it is impossible to determine whether a reported error (from Send_Error) was encountered because of the received data or after the processing of the change of direction.

This situation is not a problem when using LU 6.2 verbs because the information is received separately — first the data, then the change of direction indicator — and there is no ambiguity about where an error might have occurred.

The ambiguity is eliminated in CPI Communications by adding the **Send-Pending** state and *error_direction* characteristic. CPI Communications places the program in **Send-Pending** state whenever the program has received data and a *status_received* parameter of CM_SEND_RECEIVED (indicating a change of direction). Then, if the program encounters an error, it uses the Set_Error_Direction call to indicate how the error occurred. If the program is in **Send-Pending** state and issues a Send_Error, CPI Communications examines the *error_direction* characteristic and notifies the partner program accordingly:

- If *error_direction* is set to CM_RECEIVE_ERROR, the partner program receives a *return_code* of CM_PROGRAM_ERROR_PURGING. This indicates that the error at the remote program occurred in the data, before (in LU 6.2 terms) the change-direction indicator was received.
- If *error_direction* is set to CM_SEND_ERROR, the partner program receives a *return_code* of CM_PROGRAM_ERROR_NO_TRUNC. This indicates that the error at the remote program occurred in the send processing after the change-direction indicator was received.

For an example of how CPI Communications uses the **Send-Pending** state and the *error_direction* characteristic, see "Example 7: Error Direction and Send-Pending State" on page 38.

Can CPI-Communications Programs Communicate with APPC Programs?

Programs written using CPI Communications can communicate with APPC programs. Some examples of the limitations on the APPC program are:

- CPI Communications does not support PIP data.
- CPI Communications does not allow the specification of MAP_NAME.
- CPI Communications does not allow the specification of FMH_DATA.
- APPC programs with names containing characters no longer allowed may require a name change. See "SNA Service Transaction Programs" for a discussion of service transaction programs.

Note: Programs written using LOCKS = LONG will work because this optimization is wholly contained, on the receiving side of the conversation, in the half-session component of the LU. However, if an APPC program requires that its partner CPI-Communications program make use of LOCKS = LONG, that function will not be supported because the CPI-Communications program has no way of specifying LOCKS = LONG.

SNA Service Transaction Programs

If a CPI-Communications program wishes to specify an SNA service transaction program, the character set shown in Appendix A, "Variables and Characteristics" for *TP_name* is inadequate. The first character of an SNA service transaction program name is a character ranging in value from X'00' through X'0D' and X'10' through X'3F' (excluding X'0E' and X'0F'). Refer to *SNA Transaction Programmer's Reference Manual for LU Type 6.2* for more details on SNA service transaction programs.

A CPI-Communications program that has the appropriate privilege may specify the name of an SNA service transaction program for its partner *TP_name*. **Privilege** is an identification that a product or installation defines in order to differentiate LU services transaction programs from other programs, such as application programs. *TP_name* cannot specify an SNA service transaction program name at the mapped conversation protocol boundary.

How to Use the Table

Table 8 beginning on page 150 is set up as a table with LU 6.2 verbs and their parameters on the left side and CPI-Communications calls across the top. The table relates a verb or verb parameter to a call (not a call to a verb). A letter at the intersection of a verb row and parameter call column is interpreted as follows:

D This parameter has been set to a default value by the CPI-Communications call. Default values can be found in the individual call descriptions.

X A similar or equal function for the LU 6.2 verb or parameter is available from the CPI Communications call. If more than one X appears on a line for a verb, the function is available by issuing a combination of the calls.

S This parameter can be set using the CPI Communications call.

Table 8 (Page 1 of 3). Relationship of LU 6.2 Verbs to CPI-Communications Calls

LU 6.2 Verbs	CPI Communications Calls: Starter Set						Advanced Function																						
	Accept_Conv	Allocate	Deallocate	Init_Conv	Receive	Send_Data	Confirm	Confirmed	Extr_Conv_Type	Extr_Mode_Name	Extr_Part_LU_Name	Extr_Sync_Level	Flush	Prep_To_Receive	Req_To_Send	Send_Error	Set_Conv_Type	Set_Deallocate_Type	Set_Error_Direction	Set_Fill	Set_Log_Data	Set_Mode_Name	Set_Part_LU_Name	Set_Prep_To_Rec_Type	Set_Receive_Type	Set_Return_Control	Set_Send_Type	Set_Sync_Level	Set_TP_Name	Test_Req_To_Snd_Rcv
MC_ALLOCATE		X		X													X					X	X			X		X	X	
LU_NAME				D																			S							
MODE_NAME				D																		S								
TPN				D																								S		
RETURN_CONTROL				D																						S				
SYNC_LEVEL				D																							S			
SECURITY																														
PIP																														
MC_CONFIRM							X																							
REQ_TO_SEND_RECEIVED							X																							
MC_CONFIRMED								X																						
MC_DEALLOCATE			X															X												
TYPE				D														S												
MC_FLUSH													X																	
MC_GET_ATTRIBUTES										X	X	X																		
OWN_FULL_QUAL_LU_NAME																														
PARTNER_LU_NAME											X																			
PART_FULL_QUAL_LU_NAME											X																			
MODE_NAME										X																				
SYNC_LEVEL												X																		
SECURITY_USER_ID																														
SECURITY_PROFILE																														
LUW_IDENTIFIER																														
CONV_CORRELATOR																														
MC_POST_ON_RECEIPT																														
MC_PREPARE_TO_RECEIVE														X																
TYPE				D																				S						
LOCKS																														
MC_RECEIVE_AND_WAIT					X																									
REQ_TO_SEND_RECEIVED					X																									
WHAT_RECEIVED					X																									
MAP_NAME																														
MC_RECEIVE_IMMEDIATE					x																									
REQ_TO_SEND_RECEIVED					X																									
WHAT_RECEIVED					X																									
MAP_NAME																														
MC_REQUEST_TO_SEND															X															

Table 8 (Page 2 of 3). Relationship of LU 6.2 Verbs to CPI-Communications Calls

LU 6.2 Verbs	CPI Communications Calls: Starter Set						Advanced Function																					
	Accept_Conv	Allocate	Deallocate	Init_Conv	Receive	Send_Data	Confirm	Confirmed	Extr_Conv_Type	Extr_Mode_Name	Extr_Part_LU_Name	Extr_Sync_Level	Flush	Prep_To_Receive	Req_To_Send	Send_Error	Set_Conv_Type	Set_Deallocate_Type	Set_Error_Direction	Set_Fill	Set_Log_Data	Set_Mode_Name	Set_Part_LU_Name	Set_Prep_To_Rec_Type	Set_Receive_Type	Set_Return_Control	Set_Send_Type	Set_Sync_Level	Set_TP_Name	Test_Req_To_Snd_Rcv
MC_SEND_DATA						X																							
MAP_NAME																													
FMH_DATA																													
REQ_TO_SEND_RECEIVED						X																							
MC_SEND_ERROR																X													
REQ_TO_SEND_RECEIVED																X													
MC_TEST																													X
TEST=POSTED																													
TEST=REQ_TO_SND_RCVD																													X
BACKOUT																													
GET_TYPE									X																				
SYNCPT																													
REQ_TO_SEND_RECEIVED																													
WAIT																													
RESOURCE_POSTED																													
ALLOCATE		X		X													X					X	X			X		X	X	
LU_NAME				D																			S							
MODE_NAME				D																		S								
TPN				D																									S	
TYPE				D													S													
RETURN_CONTROL				D																						S				
SYNC_LEVEL				D																							S			
SECURITY																														
PIP																														
CONFIRM							X																							
REQ_TO_SEND_RECEIVED							X																							X
CONFIRMED								X																						
DEALLOCATE			X															X			X									
TYPE				D														S												
LOG_DATA				D																	S									
FLUSH													X																	

Table 8 (Page 3 of 3). Relationship of LU 6.2 Verbs to CPI-Communications Calls

LU 6.2 Verbs	*CPI Communications Calls*																														
	Starter Set							**Advanced Function**																							
	Accept_Conv	Allocate	Deallocate	Init_Conv	Receive	Send_Data	Confirm	Confirmed	Extr_Conv_Type	Extr_Mode_Name	Extr_Part_LU_Name	Extr_Sync_Level	Flush	Prep_To_Receive	Req_To_Send	Send_Error	Set_Conv_Type	Set_Deallocate_Type	Set_Error_Direction	Set_Fill	Set_Log_Data	Set_Mode_Name	Set_Part_LU_Name	Set_Prep_To_Rec_Type	Set_Receive_Type	Set_Return_Control	Set_Send_Type	Set_Sync_Level	Set_TP_Name	Test_Req_To_Snd_Rcv
GET_ATTRIBUTES										X	X	X																			
OWN_FULL_QUAL_LU_NAME																															
PARTNER_LU_NAME											X																				
PART_FULL_QUAL_LU_NAME											X																				
MODE_NAME										X																					
SYNC_LEVEL												X																			
SECURITY_USER_ID																															
SECURITY_PROFILE																															
LUW_IDENTIFIER																															
CONV_CORRELATOR																															
POST_ON_RECEIPT																															
FILL																															
PREPARE_TO_RECEIVE														X										X							
TYPE				D																				S							
LOCKS																															
RECEIVE_AND_WAIT					X															X					X						
FILL				D																S											
REQ_TO_SEND_RECEIVED					X																										
WHAT_RECEIVED					X																										
RECEIVE_IMMEDIATE					X															X					X						
FILL				D																S											
REQ_TO_SEND_RECEIVED					X																										
WHAT_RECEIVED					X																										
REQUEST_TO_SEND															X																
SEND_DATA						X																									
REQ_TO_SEND_RECEIVED						X																									
SEND_ERROR																X			X		X										
TYPE																															
LOG_DATA																D					S										
REQ_TO_SEND_RECEIVED																X															
TEST																															X
TEST = POSTED																															
TEST = REQ_TO_SND_RCVD																															X

Appendix E. CMS VM/SP—Extension Information

This appendix contains information about VM/SP extensions to CPI Communications. This extension information consists of:

- Additional VM/SP-related notes about CPI-Communications routines
- Special routines that can be used in CMS to take advantage of VM/SP's capabilities. However, note that a program using any of these VM/SP extension routines cannot be moved to another system without being changed.
- Programming language notes for writing CPI-Communications programs in VM/SP.

VM/SP readers should also be familiar with chapters 1-4 of the *VM/SP Connectivity Programming Guide and Reference*, SC24-5377. That VM/SP book discusses connectivity terminology, gives an overview of communication programming on VM/SP, and introduces CPI Communications. It also contains scenarios and example programs that aid understanding of CPI Communications in VM/SP.

Invoking CPI-Communications Routines in VM/SP

In addition to the SAA languages shown in "Call Syntax" on page 42, CPI-Communications routines in VM/SP can be called from Pascal and PL/I. Here is the calling format for these two languages:

Pascal

```
routine_name (parm0,parm1,parm2,...parmN);
```

PL/I

```
CALL routine_name (parm0,parm1,parm2,...parmN);
```

If a VM/SP program cannot successfully invoke the CPI-Communications routine it is trying to call, the following error message is generated:

```
1292E   Error calling CPI Communications routine, return code=retcode
```

The possible values for *retcode* in this message, and their meanings, are as follows:

Code	Meaning
-07	The CPI-Communications routine that was called was not loaded. Issue the following command: `'RTNLOAD * (FROM VMLIB SYSTEM GROUP VMLIB)'` and then try calling the routine again. If this fails, contact the system administrator.
-08	The CPI-Communications routine that was called has been dropped. Follow the same steps as for return code -07.
-09	Insufficient storage is available. Obtain more storage.
-10	Too many parameters were specified for the CPI-Communications routine. Refer to the call's detailed description to find the proper number of parameters.

-11 Not enough parameters were specified for the CPI-Communications routine. Follow the same step as for return code -10.

If one of these error condition occurs, a user abend is generated.

Special VM/SP Notes

There are some special considerations that should be understood when writing applications for a VM/SP environment. These are explained in this section.

Side Information

CPI Communications defines side information, which is a set of predefined values used when starting conversations. The side information consists of a partner LU name, mode name, and TP name, and it is indexed by a symbolic destination name.

VM/SP implements side information using CMS communications directory files. A communications directory file is a NAMES file that can be set up on a system level (by a system administrator) or on a user level. A CMS communications directory can contain the following tags:

Table 9. Contents of a CMS Communications Directory File

Tag	What the Value on the Tag Specifies
:nick.	Symbolic destination name for the target resource.
:luname.	The partner LU name (locally known LU name) that identifies where the resource resides. This name consists of two 8-byte fields: 1. An LU_name_qualifier, which is either: • *USERID (for connections to private resources within the TSAF collection), • *IDENT or blank (for connections to local or global resources within the collection), or • a defined gateway name (for connections outside the collection). 2. A *target_LU_name* that specifies either: • The target virtual machine's user ID (for connections to private resources within the collection), • Zero (for connections to local or global resources within the collection), or • The name of the partner's LU (for connections outside the collection).
:tpn.	The target resource ID (transaction program name).
:modename.	For connections outside the TSAF collection, this field specifies the mode name for the SNA session connecting the gateway (a TSAF collection) and the target LU. For connections within the TSAF collection, this field specifies a mode name of either VMINT or VMBAT, or it is omitted.
:security.	The security type of the conversation (NONE, SAME, or PGM). Currently, only security types NONE and PGM are supported for connections out of a TSAF collection.
:userid.[1]	The access security user ID.
:password.[1]	The access security password.

Once the communications directory file is created, it is put into effect by using the SET COMDIR command. (Refer to the *VM/SP CMS Command Reference* for details on this command.)

Note: If, when running a program, program execution must be halted by use of the HX command, any communication directory files that were loaded with SET COMDIR are unloaded. If this happens, the SET COMDIR RELOAD command must be reissued to get the communications directory file loaded again.

Interrupts

For CPI Communications to work properly in VM/SP, external interrupts must be enabled for a user's virtual machine. Any action taken by a user application to disable interrupts must be done carefully.

VM/SP-Specific Errors

Return codes for CPI-Communications routines are listed with each routine in Chapter 4, and they are generically described in Appendix B; however, calls to CPI Communications in VM/SP can produce return codes for VM/SP-specific reasons. The following list shows some CPI Communications return codes, along with some possible VM/SP-specific causes:

CM_RESOURCE_FAILURE_NO_RETRY
: This code can result when the partner program did one of the following:

  - Issued the Terminate_Resource_Manager routine (XCTRRM).
  - Re-IPLed CMS or logged off.

CM_SECURITY_NOT_VALID
: This code can result when the user ID trying to allocate a conversation to a private resource is not authorized in the private server's $SERVER$ NAMES file.

CM_TP_NOT_AVAIL_NO_RETRY
: This code can result when the connection to the target cannot be completed due to one of the following:

  - The program does not have directory authorization to allocate the conversation.
  - The program tried allocating a conversation to a private resource manager program, but the private server virtual machine either had SET SERVER OFF or SET FULLSCREEN ON.
  - The target server virtual machine has exceeded its maximum number of connections.

CM_TPN_NOT_RECOGNIZED
: This code can result when the connection to the target cannot be completed due to one of the following:

  - The target local or global server virtual machine has not issued the Identify_Resource_Manager routine (XCIDRM).

---

[1] Access security user IDs and passwords can be specified on the APPCPASS statement in the source virtual machine's directory, rather than in this file. The *VM/SP Connectivity Planning, Administration, and Operation* book explains this in detail.

- The target private server virtual machine cannot be autologged.
- For private resource manager programs, the resource ID was not registered in the private server's $SERVER$ NAMES file.
- The CMS-invoked routine in a private server (specified on the :module. tag in $SERVER$ NAMES) is unknown.

CPI Communications defines a return code called CM_PRODUCT_SPECIFIC_ERROR for each routine. In VM/SP, whenever a call to a CPI-Communications routine results in this return code, a file called CPICOMM LOGDATA A is appended with a message line that describes the cause of the error. The following list shows the possible VM/SP messages associated with the CM_PRODUCT_SPECIFIC_ERROR return code:

Accept_Conversation (CMACCP)

- Unable to get storage.
- HNDIUCV SET failed.

Allocate (CMALLC)

- The partner program can not be on the same virtual machine as that of the program issuing the Allocate.
- Unable to set alternate userid
- Unable to get storage

Initialize_Conversation (CMINIT)

- Unable to get storage
- Bad Side-Information Security value

Receive (CMRCV)

- Unable to get storage
- APPC/VM RECEIVE returned neither data nor status

Set_Partner_LU_Name (CMSPLN)

- The partner LU name in VM/SP cannot contain a period.

Other VM/SP-Specific Notes for CPI-Communications Routines

This section contains notes that the programmer should be aware of when using CPI-Communications routines in VM/SP.

Allocate (CMALLC)

- If the target program is within the same TSAF collection of VM/SP systems, an LU 6.2 session does not have to be allocated.

Set_Partner_LU_Name (CMSPLN)

- In VM/SP, the partner LU name is often referred to as a "locally known LU name," consisting of these two entities:
  1. An *LU_name_qualifier*, which is either:
     - *USERID (for connections to private resources within the TSAF collection)
     - *IDENT or blank (for connections to local or global resources within the collection), or
     - A defined gateway name (for connections outside the collection).

2. A *target_LU_name* that specifies either:
   - The target virtual machine's user ID (for connections to private resources within the collection),
   - Zero (for connections to local or global resources within the collection), or
   - The name of the partner LU (for connections outside the collection).

- Exactly one blank should be used to separate the two partner LU name fields.
- VM/SP allows the partner LU specified to be local (within the initiating LU); the partner LU does not have to be remote.

Extract_Partner_LU_Name (CMEPLN)

- The partner LU name in VM/SP (sometimes called locally known LU name) extracted consists of two fields: an LU name qualifier and a source LU name. The partner LU name returned by this routine always has a blank separating the two fields.

Overview of VM/SP Extension Routines

As mentioned earlier, VM/SP provides a number of routines that are extensions to SAA CPI Communications. Programs using these routines may not be portable to other SAA systems. However, these routines can be used to take advantage of the VM/SP operating system. These extension routines are grouped into the three sections that follow.

Security

The default security value for CPI-Communications conversations is SAME. VM/SP provides a routine called Set_Conversation_Security_Type that allows explicit specification of the security value (NONE, SAME, or PROGRAM) for the conversation.

In addition, VM/SP provides routines that allow explicit specification of an access security user ID (Set_Conversation_Security_User_ID) and an access security password (Set_Conversation_Security_Password).

Resource Manager Programs

Using CPI Communications, servers can only accept a single conversation and then finish. VM/SP provides routines that allow resource manager programs in server virtual machines to handle more than one conversation. The routines are called Identify_Resource_Manager, its counterpart Terminate_Resource_Manager, and Wait_on_Event.

Intermediate Servers

Using CPI Communications, a server can only be on the accepting side of a conversation; it cannot allocate conversations. VM/SP provides a routine that a server program can use before making a connection on behalf of another program. This routine, called Set_Client_Security_User_ID, explicitly sets the user ID of the program that makes the original connection request. In this way, the original requesting program is identified to the final target program.

Summary

The following table summarizes VM/SP routines that are extensions to CPI Communications. The routines are listed in alphabetical order by their callable name. The last column of the table shows the page where the routine is described in detail.

Table 10 (Page 1 of 2). Overview of VM/SP Extension routines

Call	Pseudonym	Description	Page
XCECSU	Extract_Conversation_Security_User_ID	Lets a server extract the user ID associated with an incoming conversation.	160
XCIDRM	Identify_Resource_Manager	Declares to CMS a name (resource ID) by which the resource manager application will be known. For a local resource manager, this routine makes the name known to the system; for a global resource manager, this routine also makes the name known to the TSAF collection of VM/SP systems.	162
XCSCSP	Set_Conversation_Security_Password	Sets the security password value for the conversation. The target LU uses this value and the security user ID to verify the identity of the requester.	165
XCSCST	Set_Conversation_Security_Type	Sets the security level for the conversation. The security level determines what security information is sent to the target. This lets the target verify the identity of the requester.	167
XCSCSU	Set_Conversation_Security_User_ID	Sets the security user ID value for the conversation. The target LU uses this value and the security password to verify the identity of the requester.	169

Table 10 (Page 2 of 2). Overview of VM/SP Extension routines

Call	Pseudonym	Description	Page
XCSCUI	Set_Client_Security_User_ID	Lets an intermediate server initiate a conversation on behalf of a specific client application. A program issuing this routine must have special authorization.	171
XCTRRM	Terminate_Resource_Manager	Ends ownership of a resource by a resource manager program.	173
XCWOE	Wait_on_Event	Allows an application to wait on communications from one or more partners. Events posted are allocation requests, information input, notification that resource management has been revoked, and console input.	174

Extract_Conversation_Security_User_ID (XCECSU)

Use the Extract_Conversation_Security_User_ID routine (XCECSU) in a program to extract the user ID associated with an incoming conversation.

The incoming conversation must have a *conversation_security_type* of XC_SECURITY_SAME or XC_SECURITY_PROGRAM. If the conversation security type is XC_SECURITY_NONE, this variable will contain nulls (X'00's), and the length is set to zero.

This routine is valid only after the server has accepted the conversation, using the Accept_Conversation routine (CMACCP).

Format

```
CALL XCECSU(conversation_id,
            conversation_security_user_id,
            conversation_security_user_id_length,
            return_code)
```

Parameters

conversation_id
Specifies the conversation identifier. This variable must be an eight-byte character string, and it is used as input to the routine.

conversation_security_user_id
Specifies the user ID obtained by this routine and returned to the calling program. This variable must be an eight-byte character string.

conversation_security_user_id_length
Specifies the length, in bytes, of the returned *conversation_security_user_ID*. This variable must be declared a four-byte signed integer with a value from 0 to 8.

return_code
Specifies the return code that is passed back from the communications routine to the calling program. This return code variable must be a signed, four-byte binary integer. The *return_code* variable can have one of the following values:

- CM_OK
  Normal completion.
- CM_PRODUCT_SPECIFIC_ERROR
  An internal control block problem prevented Extract_Conversation_Security_User_ID from getting the user ID. When this code is returned, a file named CPICOMM LOGDATA A is appended with the following line:

  ```
  CPIC block does not exist
  ```

- CM_PROGRAM_PARAMETER_CHECK
  The specified conversation ID was not found.

State Changes

This routine does not cause any state changes.

Identify_Resource_Manager (XCIDRM)

Use the Identify_Resource_Manager (XCIDRM) routine to declare to CMS the name of a resource that the application wants to manage. How the resource name is known depends on the type of application being written:

- For a local resource manager application, the resource name is known to the system.
- For a global resource manager application, the resource name is known to the local system and the entire TSAF collection of VM/SP systems.
- For a private resource manager application, the resource name is known only to the virtual machine where the program is running.

This routine also specifies how the application handles incoming conversations. See description of *service mode* below.

Format

```
CALL XCIDRM(resource_id,
            resource_manager_type,
            service_mode,
            security_level_flag,
            return_code)
```

Parameters

resource_ID
Specifies the name of a resource managed by this resource manager application. This variable must be an eight-byte character string, padded on the right with blanks if necessary. It is used as input to the routine.

Other applications wishing to use this resource name must establish conversations to this resource. These allocation requests are then routed to the application that issued this Identify_Resource_Manager routine.

resource_id corresponds to the transaction program name that requesting applications supply when allocating a conversation to this resource.

resource_manager_type
Identifies whether the application is a private, local, or global resource manager. *resource_manager_type* must be equal to one of the following values:

- XC_PRIVATE
  Private resource names are identified only to the virtual machine in which they are active, but they can be accessed by programs that have the proper authorization. These other programs can reside on the same VM/SP system, same TSAF collection of VM/SP systems, or other system in an SNA network.
- XC_LOCAL
  Local resource names are identified only to the system in which they reside, and cannot be accessed from outside this system.
- XC_GLOBAL
  Global resource names are identified to an entire TSAF collection. They may be accessed by other programs in the collection, or in an SNA network.

This parameter is used as input to the routine.

service_mode
Indicates how this resource manager application handles conversations. *service_mode* must be equal to one of the following values:

- XC_SINGLE
  This resource manager program can accept only a single conversation. When the single conversation is ended (deallocated), the resource manager program should be written to issue the Terminate_Resource_Manager (XCTRRM) for *resource_id*.

  If a program makes an allocation request to a private resource name and the private resource manager program has already accepted a conversation for that resource name, CMS queues the new request; then when the private resource manager program ends (after the single conversation is deallocated and the program issues XCTRRM), CMS automatically restarts the private resource manager program and takes the first pending request off of the queue.
- XC_SEQUENTIAL
  This resource manager program can accept only one conversation at a time. However, when one conversation is completed and deallocated, the resource manager program can issue Wait_on_Event (XCWOE) to wait for the next allocation request, or issue Accept_Conversation (CMACCP). (If a program issues CMACCP and no allocations are pending, the program abends, however.) The program can continue in this sequence until all conversations are completed, and then it should issue Terminate_Resource_Manager for the resource ID.

  If a program makes an allocation request to a private resource name and the private resource manager program has already accepted a conversation for that resource name, CMS queues the new request; then after the single conversation is deallocated, the program should issue XCWOE or CMACCP as described in the preceding paragraph.
- XC_MULTIPLE
  This resource manager program can accept multiple, simultaneous conversations. A "multiple mode" program can issue Wait_on_Event (XCWOE) for more than one conversation at a time.

  If a program makes an allocation request to a private resource name and the private resource manager program has already accepted a conversation for that resource name, CMS queues the new request. However, in this case, the program does not have to deallocate the previous conversation for a new allocation request to be presented.

For all three cases, if a program makes an allocation request to a global or local resource name and the resource manager program has already accepted a conversation for that resource name, the source program gets a "deallocated" indication.

This parameter is used as input to the routine.

security_level_flag
Indicates whether or not this resource manager will accept inbound connections that have *conversation_security_type* equal to XC_SECURITY_NONE. *security_level_flag* must be equal to one of the following values:

- XC_REJECT_SECURITY_NONE
  The resource manager will not accept connections that have *conversation_security_type* equal to XC_SECURITY_NONE. The source program requesting such a connection will get *return_code* of CM_ALLOCATE_FAILURE_NO_RETRY on its Allocate (CMALLC) routine.
- XC_ACCEPT_SECURITY_NONE
  The resource manager will accept connections that have *conversation_security_type* equal to XC_SECURITY_NONE.

This parameter is used as input to the routine.

return_code
Specifies the return code that is passed back from the communications routine to the calling program. This return code variable must be a signed, four-byte binary integer. The *return_code* variable can have one of the following values:

- CM_OK
  Normal completion.
- CM_PRODUCT_SPECIFIC_ERROR
  A storage failure prevented the resource manager program from being identified. When this code is returned, a file named CPICOMM LOGDATA A is appended with the following line:

  ```
  Unable to get storage
  ```

- CM_PROGRAM_PARAMETER_CHECK
  This can result from one of the following conditions:
  - *resource_id* has already been declared to CMS
  - *resource_id* could not be declared to CMS
  - *resource_manager_type* contains an invalid value
  - *service_mode* contains an invalid value
  - *security_level_flag* contains an invalid value.
- CM_UNSUCCESSFUL
  Identify_Resource_Manager was unable to get ownership of the resource because it is already owned by another virtual machine. This return code applies only when *resource_manager_type* is XC_LOCAL or XC_GLOBAL.

State Changes

This routine is not specific to a conversation, so it does not cause any state changes.

Usage Notes

1. The application does not need to call Identify_Resource_Manager if:
   - The application initiates all its conversations and is never the target of an allocation request.
   - The application is a private resource manager, invoked by CMS as the target of a single conversation.
2. An application calling Identify_Resource_Manager should also call the Terminate_Resource_Manager routine (XCTRRM) before exiting.

Set_Conversation_Security_Password (XCSCSP)

Use the Set_Conversation_Security_Password routine (XCSCSP) in a source program or intermediate server to set the security password for a conversation. The password is necessary for a connection request that has a *conversation_security_type* of XC_SECURITY_PROGRAM.

Set_Conversation_Security_Password can only be issued for a conversation that is in **Initialize** state. It cannot be issued at any point following an Allocate routine (CMALLC).

Format

```
CALL XCSCSP(conversation_id,
            conversation_security_password,
            conversation_security_password_length,
            return_code)
```

Parameters

conversation_id
Specifies the conversation identifier. This variable must be an eight-byte character string, and it is used as input to the routine.

conversation_security_password
Specifies the password. The target LU uses this value and the user ID to verify the identity of the source program making the allocation request. The password is stored temporarily in the LU's conversation control block; it is erased at completion of an Allocate routine.

conversation_security_password must be declared as an eight-byte character string, and it is used as input to the routine.

conversation_security_password_length
Specifies the length of the security password. This must be a four-byte binary integer value from 0 to 8, and it is used as input to the routine.

return_code
Specifies the return code that is passed back from the communications routine to the calling program. This return code variable must be a signed, four-byte binary integer. The *return_code* variable can have one of the following values:

- CM_OK
  Normal completion.
- CM_PROGRAM_PARAMETER_CHECK
  This can result from one of the following:
  - *conversation_id* specifies an unassigned conversation ID,
  - *conversation_security_password_length* specifies a value of less than 0 or greater than 8.
- CM_PROGRAM_STATE_CHECK
  The conversation is not in **Initialize** state or the *conversation_security_type* is not XC_SECURITY_PROGRAM.

State Changes

This routine does not cause any state changes.

Usage Notes

1. A program can only issue Set_Conversation_Security_Password on an outgoing conversation.
2. Before a program can issue Set_Conversation_Security_Password, a user ID must be provided. The user ID either comes from
   - The program issuing Set_Conversation_Security_User_ID (XCSCSU), or
   - A communications directory file.
3. When *conversation_security_type* is XC_SECURITY_PROGRAM, a program can issue this XCSCSP routine or have a password specified in the virtual machine's
   a. Communications directory file, or
   b. APPCPASS directory statement.

   The access security password specified on this routine overrides a password in the communications directory file and causes an access security password specified on an APPCPASS directory statement to be ignored.

Set_Conversation_Security_Type (XCSCST)

Use the Set_Conversation_Security_Type routine (XCSCST) to set the security type for the conversation. This routine overrides the value that was assigned when the conversation was initialized.

Set_Conversation_Security_Type can only be called from a program in **Initialize** state. It cannot be issued at any point following an Allocate routine (CMALLC).

Format

```
CALL XCSCST(conversation_id,
            conversation_security_type,
            return_code)
```

Parameters

conversation_id
Specifies the conversation identifier. This variable must be an eight-byte character string, and it is used as input to the routine.

conversation_security_type
Specifies the kind of access security information that the program is sending to its target. The target LU uses this security information to verify the identity of the source. The access security information, if present, consists of either a user ID or a user ID and password. *conversation_security_type* is used as input to the routine, and it must be equal to one of the following values:

- XC_SECURITY_NONE
  No access security information is to be included on the allocation request to the target resource manager.
- XC_SECURITY_SAME
  The user ID of the source program's virtual machine is sent on the allocation request to the target resource manager.

  This security type can not be used for allocations outside the program's TSAF collection.
- XC_SECURITY_PROGRAM
  The source program must supply an access user ID and password on the allocation request to the target LU.

return_code
Specifies the return code that is passed back from the communications routine to the calling program. This return code variable must be a signed, four-byte binary integer. The *return_code* variable indicates the result of the call execution and can have one of the following values:

- CM_OK
  Normal completion.
- CM_PROGRAM_PARAMETER_CHECK
  This can result from one of the following:
  - *conversation_id* specifies an unassigned conversation ID,
  - *conversation_security_type* specifies an undefined value.
- CM_PROGRAM_STATE_CHECK
  The conversation is not in **Initialize** state.

State Changes

This routine does not cause any state changes.

Usage Notes

1. A program can only issue Set_Conversation_Security_Type on an outgoing conversation.
2. A program does not need to use this routine if the default security type of XC_SECURITY_SAME is desired, or if a security type is specified in the virtual machine's communications directory file. The security type specified on this routine overrides a security type in the communications directory file.

Set_Conversation_Security_User_ID (XCSCSU)

Use the Set_Conversation_Security_User_ID (XCSCSU) routine to set the access security user ID for the conversation. A program can only specify an access security user ID when it wants to connect to a target using a *conversation_security_type* of XC_SECURITY_PROGRAM.

Set_Conversation_Security_User_ID can only be called from **Initialize** state. It cannot be issued at any point following an Allocate (CMALLC) routine.

Format

```
CALL XCSCSU(conversation_id,
            conversation_security_user_id,
            conversation_security_user_id_length,
            return_code)
```

Parameters

conversation_id
Specifies the conversation ID. This variable must be an eight-byte character string, and it is used as input to the routine.

conversation_security_user_id
Specifies the user ID. The target LU uses this value (and the *conversation_security_password*, which can be specified on the Set_Conversation_Security_Password routine) to verify the identity of the user making the allocation request. In addition, the target LU may use the user ID for auditing or accounting purposes.

This variable must be declared as an eight-byte character string, and it is used as input to the routine.

conversation_security_user_id_length
Specifies the length, in bytes, of the security user ID. This variable must be a four-byte signed integer with a value from 0 to 8.

return_code
Specifies the return code that is passed back from the communications routine to the calling program. This return code variable must be a signed, four-byte binary integer. The return code indicates the result of call execution and can have one of the following values:

- CM_OK
  Normal completion.
- CM_PROGRAM_PARAMETER_CHECK
  This can result from one of the following:
  - *conversation_id* specifies an unassigned conversation ID,
  - *conversation_security_user_id_length* specifies a value less than 0 or greater than 8.
- CM_PROGRAM_STATE_CHECK
  The conversation is not in **Initialize** state.

State Changes

This routine does not cause any state changes.

Usage Notes

1. A program can only issue Set_Conversation_Security_User_ID on an outgoing conversation.
2. When *conversation_security_type* is XC_SECURITY_PROGRAM, a program can issue this XCSCSP routine or have a user ID specified in the virtual machine's
   a. CMS communications directory file, or
   b. APPCPASS directory statement.

   The access security user ID specified on this routine overrides a user ID in the communications directory file and causes an access security user ID specified in a directory APPCPASS statement to be ignored.

Set_Client_Security_User_ID (XCSCUI)

Use the Set_Client_Security_User_ID (XCSCUI) routine in an intermediate server to set a user ID value based on an incoming conversation's user ID. The intermediate server can then present this user ID to the final target when it initiates a conversation on behalf of the client application.

A program that acts as an intermediate server might have incoming conversations from various virtual machines. With Set_Client_Security_User_ID, such a server can specify a particular user ID that will be presented to the final target resource manager. In this way, the target resource manager virtual machine knows where the original request is coming from.

A server can only call Set_Client_Security_User_ID if the following conditions are true:

- The incoming conversation has *conversation_security_type* equal to XC_SECURITY_SAME or XC_SECURITY_PROGRAM.
- The outgoing conversation from the intermediate server has *conversation_security_type* equal to XC_SECURITY_SAME.
- The intermediate server is in **Initialize** state for the outgoing *conversation_id*.
- The intermediate server virtual machine is authorized to issue a Diagnose Code X'D4' (for defining an alternate user ID) to issue an Allocate (CMALLC) on behalf of a client application. This authorization is privilege class B (unless default privilege classes have been changed).

Format

```
CALL XCSCUI(conversation_id,
            client_user_id,
            return_code)
```

Parameters

conversation_id
Specifies the conversation identifier. This variable must be an eight-byte character string, and it is used as input to the routine.

client_user_id
Identifies the client's user ID, obtained by calling the Extract_Conversation_Security_User_ID (XCECSU) routine for the conversation between the server and the client application. This variable must be an eight-byte character string, padded on the right with blanks as necessary, and it is used as input to the routine.

return_code
Specifies the return code that is passed back from the communications routine to the calling program. This return code variable must be a signed, four-byte binary integer. The *return_code* variable can have one of the following values:

- CM_OK
  Normal completion.
- CM_PROGRAM_PARAMETER_CHECK
  This can result from one of the following:
  - *conversation_id* is not found

  - *conversation_security_type* for the incoming conversation is not equal to XC_SECURITY_SAME or XC_SECURITY_PROGRAM.
- CM_PROGRAM_STATE_CHECK
  The conversation is not in **Initialize** state.

State Changes

This routine does not cause any state changes.

Usage Notes

1. Here is a typical sequence of events that includes an intermediate server calling the Set_Client_Security_User_ID:
   a. A client application allocates a conversation with the server application.
   b. The server application accepts the conversation using the Accept routine (CMACCP).
   c. The server application calls the Extract_Conversation_Security_User_ID routine (XCECSU), on the conversation with the client application, to get the client's user ID.
   d. The server calls Initialize_Conversation (CMINIT) to get a conversation ready to allocate on behalf of the client.
   e. The server application calls Set_Client_Security_User_ID using the extracted user ID, to set the security information of the new conversation.
   f. The intermediate server application calls Allocate (CMALLC) for the conversation that is being initialized on behalf of the client application. (The default security type of XC_SECURITY_SAME is used on this CMALLC call.)
   g. The final target program is presented with the original source program's user ID.
2. An intermediate server that does not use Identify_Resource_Manager (XCIDRM) always uses the source program's user ID when allocating a conversation to the final target; such an intermediate server does not need to use Set_Client_Security_User_ID.
3. An intermediate server that issues Identify_Resource_Manager and does not use Set_Client_Security_User_ID will forward its own user ID, not the original source program's, when allocating a conversation to the final target.

Terminate_Resource_Manager (XCTRRM)

Use the Terminate_Resource_Manager routine (XCTRRM) from a resource manager application to end management of a resource. The resource manager automatically ends all conversations for the specified resource ID, and the communication control blocks used to manage the resource are released.

If the resource ID specified on this routine is a global or local resource, that name is no longer identified to the TSAF collection of VM/SP systems.

Format

```
CALL XCTRRM(resource_id,
            return_code)
```

Parameters

resource_id
Specifies the name of a resource, managed by this resource manager application, for which service is being terminated. This is a name that was specified by this application on a previous call to the Identify_Resource_Manager routine (XCIDRM). *resource_id* is a character string variable eight bytes in length, and it is used as input to the routine.

return_code
Specifies the return code that is passed back from the communications routine to the calling program. This return code variable must be a signed, four-byte binary integer. The *return_code* variable can have one of the following values:

- CM_OK
  Normal completion.
- CM_PROGRAM_PARAMETER_CHECK
  This virtual machine does not control the specified resource.

State Changes

Reset state is entered on all conversations associated with the specified *resource_id*. (This applies when *return_code* is CM_OK, indicating normal completion.)

Usage Note

1. Any resource manager application that called the Identify_Resource_Manager routine should be sure to call Terminate_Resource_Manager before exiting.

   Failure to call either Terminate_Resource_Manager to deallocate all conversations could result in paths and control blocks remaining active within the virtual machine.

Wait_on_Event (XCWOE)

Use the Wait_on_Event routine (XCWOE) from a source or target application to wait for communications from one or more partners. This routine provides a way to handle requests from partner programs; a program can issue Wait_on_Event, then take action according to the type of request it receives.

Format

```
CALL XCWOE(resource_id,
           conversation_id,
           event_type,
           data_length,
           console_input_buffer,
           return_code)
```

Parameters

resource_id
Specifies the name of a resource managed by this resource manager application for which an event has been posted. *resource_id* is returned to the caller; it is a name that was specified by this application on a previous call to Identify_Resource_Manager (XCIDRM). It must be a character string variable eight bytes in length, and it is passed back as output from the routine.

This parameter is only valid when the posted event is an allocation request or a revoke resource notification. (If the event is information input or console input, the contents of this variable should not be examined.)

conversation_id
Identifies the conversation on which data is available to be received. *conversation_id* is returned to the caller as output from the routine. It must be a character string variable eight bytes in length.

This parameter is only valid when the event is information input. (If the event is an allocation request, revoke resource notification, or console input, the contents of this variable should not be examined.)

event_type
Indicates the type of event posted. This variable is used as output from the routine, and it must be equal to one of the following values:

- XC_ALLOCATION_REQUEST
  A source program is attempting to allocate a conversation with the program. The program must issue an Accept_Conversation (CMACCP) to clear this posted event.
- XC_INFORMATION_INPUT
  A partner program is attempting to communicate information to the program. For instance, it might be sending data or deallocating its connection. The program must issue a Receive (CMRCV) to clear this posted event.
- XC_RESOURCE_REVOKED
  Another program has revoked the program's resource. In this case, the program's resource manager program must issue a Terminate_Resource_Manager (XCTRRM) when it completes its active conversation(s).

- XC_CONSOLE_INPUT
  The program is waiting for information from the console. This information will go away after it is presented to the program via the *console_input_buffer*.

data_length
If the posted event is information input or console input, this indicates the amount of data bytes that are available to be received. See Usage Note 3 on page 176. If the posted event is console input, this indicates how many bytes of data are being sent. *data_length* must be a four-byte integer variable, and it is used as output from the routine.

This parameter is only valid when the event is information input or console input. (If the event is an allocation request or revoke resource notification, the contents of this variable should not be examined.)

console_input_buffer
Specifies the name of a buffer for console input. On console input events, the contents of the terminal input buffer will be stored here. This must be a character string variable, 130 bytes in length, and it is used as output from the routine. If more than 130 bytes were supplied, the data is truncated and the program will get only the first 130 bytes.

This parameter is only valid when the event is console input. (If the event is an allocation request, revoke resource notification, or information input, the contents of this variable should not be examined.)

return_code
Specifies the return code that is passed back from the communications routine to the calling program. This return code variable must be a signed, four-byte binary integer. The *return_code* variable can have one of the following values:

- CM_OK
  Normal completion.
- CM_PRODUCT_SPECIFIC_ERROR
  A storage failure prevented Wait_on_Event from being invoked. When this code is returned, a file named CPICOMM LOGDATA A is appended with the following line:

  ```
  Unable to get storage
  ```

- CM_PROGRAM_STATE_CHECK
  No conversations exist or no resource has been identified.

State Changes

This routine does not cause any state changes.

Usage Notes

1. If a resource manager application was identified as a multiple server, it can issue Wait_on_Event routines for more than one conversation at a time; if the application was identified as a single or sequential server, it may issue only Wait_on_Event routines for one conversation at a time.
2. Events are posted in this order of priority:
   a. Allocation request
   b. Information input
   c. Resource revoke notification
   d. Console input.

For example, as long as there are allocation events pending, they will be serviced first.

- For **allocation requests**:

  After receiving the allocation event indication, the application can call the Accept_Conversation routine (CMACCP) to establish the conversation and to get a *conversation_id* assigned by the system.

  A new conversation is established after the resource manager application calls the Accept_Conversation routine. If necessary, the resource manager application can get information about the conversation by calling the appropriate "Extract" routines.

  After the resource manager application accepts the conversation, it is in receive state for this conversation. It may call one of the following routines:

  - Receive, to get any data that was sent.
  - Wait_on_Event, to wait for additional communication
  - Deallocate, with *deallocate_type* set to CM_DEALLOCATE_ABEND, to terminate the conversation.

- For **information input**:

  After getting this event, the application should issue the Receive (CMRCV) routine, using the value in *data_length* returned by Wait_on_Event.

- For **resource revoked notification**:

  No new connections may be made to the specified *resource_id*. This resource ID is no longer known to the rest of the system or TSAF collection. Existing conversations are not affected. The actions taken for this event are application-specific. The resource manager still must issue Terminate_Resource_manager.

- For **console input**:

  The application determines how to interpret what is input from the console and what further action to take.

3. The *data length* parameter is a measure of the amount of data available when the event is information input or console input. When it is information input, the value of *data length* can be used on a subsequent call to the Receive routine to receive the data

   Using this length on the call to Receive will not guarantee that the Receive will complete immediately; the only way to guarantee that a Receive will complete immediately is to set *receive type* = CM_RECEIVE_IMMEDIATE prior to calling the Receive.

Programming Language Considerations

The following languages can be used on VM/SP to call CPI-Communications routines and the VM/SP extension routines:

- C
- COBOL
- CSP (Application Generator)
- FORTRAN
- REXX (SAA Procedures Language),
- Pascal
- PL/I.

Specific notes for each of these languages is described later in individual sections. In addition, the following note applies to all the languages:

- Prior to running the program, the following CMS commands must be issued in order to establish proper linkage with the CPI-Communications routines:

```
GLOBAL TXTLIB CMSSAA
LOAD programname (AUTO
```

C

The following notes apply to C programs using CPI-Communications routines:

- The pseudonym file CMC COPY contains C statements that allow the use of symbolic names (pseudonyms) for various CPI-Communications values.
- Before calling a CPI-Communications routine, use the extern statement to declare the routine as having external linkage, and fully prototype the routine's return value and arguments.
- When passing an integer value as a parameter, the parameter name should be preceded with an ampersand (&) so that the value is passed by reference.
- To pass a parameter as a string literal, it should be surrounded with double quotes rather than single quotes.
- VM/SP does not put a terminating null byte in character strings it returns. C programs must take this into consideration.
- In addition, if the program is being compiled with the C Program Offering:
  - The #pragma statement for OS linkage is required for each CPI-Communications routine. The statement has the following format:

    ```
    #pragma linkage(routinename, OS)
    ```

  - To generate the program into a module, issue the following commands:

    ```
    LOAD C$TEXT programname DMSSAA (RESET C$START
    GENMOD filename (FROM C$TEXT)
    ```

COBOL

The following notes apply to COBOL programs using CPI-Communications routines:

- The pseudonym file CMCOBOL COPY contains COBOL statements that allow the use of symbolic names (pseudonyms) for various CPI-Communications values.

- Because COBOL does not support the underscore character (_), the underscores in COBOL pseudonyms are replaced with dashes (-). For example, COBOL programmers would use CM-IMMEDIATE as a pseudonym value name in their programs instead of CM_IMMEDIATE.
- Each argument in the parameter list must be called (listed) by name.
- Each variable in the parameter list must be level 01.
- Number variables must be fullwords (at least five but less than ten "9"s) and they must be COMP-4, not zoned decimal.

CSP (Application Generator)

The following notes apply to CSP programs using CPI-Communications routines:

- The pseudonym file CMCSP COPY explains how to use CSP data definition to define variables. It also provides sample CSP statements that initialize the symbolic variable names (pseudonyms) for various CPI-Communications values.
- Because CSP does not support the underscore character (_), the underscores in CSP pseudonyms are replaced with dollar signs ($).

FORTRAN

The following notes apply to FORTRAN programs using CPI-Communications routines:

- The pseudonym file CMFORTRN COPY contains FORTRAN statements that allow the use of symbolic names (pseudonyms) for various CPI-Communications values.
- The EXTERNAL statement should be used for each CPI Communications routine that is called.

REXX (SAA Procedures Language)

The following notes apply to REXX programs using CPI-Communications routines:

- The pseudonym file CMREXX COPY contains REXX statements that allows the use of symbolic names (pseudonyms) for various CPI-Communications values.
- Character strings returned by CPI Communications routines are stored in variables with the maximum allowable length. (Maximum lengths are shown in Appendix A and later in this Appendix.) However, there is a returned length variable associated with the returned character string, which allows use of the REXX function

  ```
  LEFT(returned_char_string,returned_length)
  ```

  to get the correct amount of data.

  (This note does not apply to returned character strings, which always have a given length. For instance, a *conversation_ID* returned from the Accept routine always has a length of 8 bytes.)

Pascal

The following notes apply to Pascal programs using CPI-Communications routines:

- A Pascal pseudonym file can be created to allow use of symbolic names (pseudonyms) for the various CPI Communications values. To create this pseudonym file, use the FORTRAN pseudonym file, CMFORTRN COPY, as a model.
- Use internal procedure statements for each CPI Communications routine that is used, and declare each routine as a FORTRAN routine.
- Parameters should be passed as variables by reference, rather than passing them as literals and constants.
- String (of char) parameters must have a length specified.
- Use PASCMOD, not LOAD, to build a load module. Follow this example to prepare the program:

```
VSPASCAL filename                    /* compile the program */
PASCMOD filename
FILEDEF OUTPUT TERM ( RECFM F LRECL 80
FILEDEF INPUT  TERM ( RECFM V LRECL 80
filename                    /* issue the name of the file to invoke it */
```

PL/I

The following notes apply to PL/I programs using CPI-Communications routines:

- A PL/I pseudonym file can be created to allow use of symbolic names (pseudonyms) for the various CPI Communications values. To create this pseudonym file, use the FORTRAN pseudonym file, CMFORTRN COPY, as a model.
- Numbers in the parameter list must be declared, initialized, and passed as variables.
- To declare each CPI Communications routine, use the following statement:

```
DCL routinename ENTRY EXTERNAL OPTIONS (ASSEMBLER INTER);
```

Variables and Characteristics

The following tables are provided for the variables and characteristics used with the VM/SP extension routines shown in this VM/SP appendix:

1. Table 11 on page 180 shows the possible values for variables and characteristics associated with VM/SP extension routines. The valid pseudonyms and corresponding integer values are provided for each variable or characteristic.
2. Table 12 on page 180 shows the data definitions for types and lengths of all VM/SP extension characteristics and variables.

These tables are extensions to the the tables provided in Appendix A, "Variables and Characteristics" on page 123. See that Appendix for further discussion of how they are used.

Table 11. VM/SP Variables/Characteristics and their Possible Values

Variable or Characteristic Name	Pseudonym Values	Integer Values
conversation_security_type	XC_SECURITY_NONE	0
	XC_SECURITY_SAME	1
	XC_SECURITY_PROGRAM	2
event_type	XC_ALLOCATION_REQUEST	1
	XC_INFORMATION_INPUT	2
	XC_RESOURCE_REVOKED	3
	XC_CONSOLE_INPUT	4
resource_manager_type	XC_PRIVATE	0
	XC_LOCAL	1
	XC_GLOBAL	2
security_level_flag	XC_REJECT_SECURITY_NONE	0
	XC_ACCEPT_SECURITY_NONE	1
service_mode	XC_SINGLE	0
	XC_SEQUENTIAL	1
	XC_MULTIPLE	2

Table 12. VM/SP Variable Types and Lengths

Variable	Variable Type	Character Set	Length (in bytes)
client_user_id	Character string	00640	8
console_input_buffer	Character string	no restriction	130
conversation_security_password	Character string	00640	8
conversation_security_password_length	Integer	N/A	4
conversation_security_type	Integer	N/A	4
conversation_security_user_id	Character string	00640	8
conversation_security_user_id_length	Integer	N/A	4
data_length	Integer	N/A	4
event_type	Integer	N/A	4
resource_id	Character string	00640	8
resource_manager_type	Integer	N/A	4
security_level_flag	Integer	N/A	4
service_mode	Integer	N/A	4

Appendix F. Sample Programs

This appendix contains the following sections:

- "SALESRPT (Initiator of the Conversation)" on page 182

  The COBOL program SALESRPT establishes a conversation with its partner program, CREDRPT, in order to transfer a sales record for credit processing. After sending the sales record, SALESRPT waits for a reply from CREDRPT.

- "CREDRPT (Acceptor of the Conversation)" on page 186

  After the conversation is started — thus causing CREDRPT to be loaded into memory and begin execution — CREDRPT accepts the conversation and receives the credit record sent by SALESRPT. When CREDRPT has successfully received the record, it sends a message back to SALESRPT informing SALESRPT of this fact.

- "Pseudonym File for COBOL" on page 191

  This section shows the COBOL statements that establish pseudonyms for the various conversation characteristic values. Both CREDRPT and SALESRPT use this file by executing the following command:

  ```
  COPY CMCOBOL.
  ```

- "Results of Successful Program Execution" on page 194

  This section shows the output generated by the

  ```
  DISPLAY
  ```

  statements in CREDRPT and SALESRPT upon successful execution of the programs.

Note: These sample programs are provided for tutorial purposes only and a complete handling of error conditions has not been shown or attempted. The details and complexity of such error handling will depend on the specific nature of actual applications.

SALESRPT (Initiator of the Conversation)

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.        SALESRPT.
*************************************************************
* THIS IS THE SALESRPT PROGRAM THAT SENDS DATA TO THE       *
* CREDRPT PROGRAM FOR CREDIT BALANCE PROCESSING.            *
*                                                           *
* PURPOSE: SEND A SALES-RECORD TO THE CREDRPT PROGRAM FOR   *
*          CREDIT BALANCE PROCESSING, THEN RECEIVE AND      *
*          DISPLAY A STATUS INDICATION FROM CREDRPT.        *
*                                                           *
* INPUT:   PROCESSING-RESULTS-RECORD FROM CREDRPT.          *
*                                                           *
* OUTPUT:  SALES-RECORD TO THE CREDRPT PROGRAM.             *
*                                                           *
*                                                           *
* NOTE:    SALES-RECORD PROCESSING HAS BEEN GREATLY         *
*          SIMPLIFIED IN THIS EXAMPLE.                      *
*************************************************************
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. IBM-370.
 OBJECT-COMPUTER. IBM-370.
 SPECIAL-NAMES.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
 I-O-CONTROL.
*
 DATA DIVISION.
 FILE SECTION.
 WORKING-STORAGE SECTION.

 01  BUFFER                    PIC  X(52)  VALUE SPACES.

 01  CM-ERROR-DISPLAY-MSG      PIC  X(40)  VALUE SPACES.

****************
* SALES-RECORD *
****************
 01  SALES-RECORD.
     05  CUST-NUM               PIC X(4)      VALUE "0010".
     05  CUST-NAME              PIC X(20)     VALUE "XYZ INC.".
     05  FILLER                 PIC X(5)      VALUE SPACES.
     05  CREDIT-BALANCE         PIC S9(7)V99  VALUE 4275.50.
     05  CREDIT-LIMIT           PIC S9(7)V99  VALUE 5000.
     05  CREDIT-FLAG            PIC X         VALUE "1".

*****************************
* PROCESSING-RESULTS-RECORD *
*****************************
 01  PROCESSING-RESULTS-RECORD   PIC X(25)  VALUE SPACES.

*********************************************
* USE THE CPI-COMMUNICATIONS PSEUDONYM FILE  *
*********************************************
     COPY CMCOBOL.
```

```
 LINKAGE SECTION.

     EJECT.
*
 PROCEDURE DIVISION.
*****************************************************************
*************************  START OF MAINLINE  *******************
*****************************************************************
 MAINLINE.

     PERFORM APPC-INITIALIZE
        THRU APPC-INITIALIZE-EXIT.
     DISPLAY "SALESRPT CONVERSATION INITIALIZED".

     PERFORM APPC-ALLOCATE
        THRU APPC-ALLOCATE-EXIT.
     DISPLAY "SALESRPT CONVERSATION ALLOCATED".

     PERFORM APPC-SEND
        THRU APPC-SEND-EXIT.
     DISPLAY "SALESRPT DATA RECORD SENT".

     PERFORM APPC-RECEIVE
        THRU APPC-RECEIVE-EXIT
        UNTIL NOT CM-OK.
     DISPLAY "SALESRPT RESULTS RECORD RECEIVED".

     PERFORM CLEANUP
        THRU CLEANUP-EXIT.
     STOP RUN.
*****************************************************************
*************************  END OF MAINLINE  *********************
*****************************************************************
*
 APPC-INITIALIZE.
     MOVE "CREDRPT" TO SYM-DEST-NAME.
************************************************
** ESTABLISH DEFAULT CONVERSATION CHARACTERISTICS **
************************************************
     CALL "CMINIT" USING CONVERSATION-ID
                         SYM-DEST-NAME
                         CM-RETCODE.
     IF CM-OK
        NEXT SENTENCE
     ELSE
        MOVE "INITIALIZATION PROCESSING TERMINATED"
             TO CM-ERROR-DISPLAY-MSG
        PERFORM CLEANUP
           THRU CLEANUP-EXIT
     END-IF.
 APPC-INITIALIZE-EXIT. EXIT.
*****************************************************************
*
 APPC-ALLOCATE.
*********************************
* ALLOCATE THE APPC CONVERSATION *
*********************************
```

```
     CALL "CMALLC" USING CONVERSATION-ID
                         CM-RETCODE
     IF CM-OK
        NEXT SENTENCE
     ELSE
        MOVE "ALLOCATION PROCESSING TERMINATED"
             TO CM-ERROR-DISPLAY-MSG
        PERFORM CLEANUP
           THRU CLEANUP-EXIT
     END-IF.
 APPC-ALLOCATE-EXIT. EXIT.
*****************************************************************
*
 APPC-SEND.
     MOVE SALES-RECORD TO BUFFER.
     MOVE 52 TO SEND-LENGTH.

***********************************
* SEND THE SALES-RECORD DATA RECORD *
***********************************
     CALL "CMSEND" USING CONVERSATION-ID
                         BUFFER
                         SEND-LENGTH
                         REQUEST-TO-SEND-RECEIVED
                         CM-RETCODE.
     IF CM-OK
        NEXT SENTENCE
     ELSE
        MOVE "SEND PROCESSING TERMINATED"
             TO CM-ERROR-DISPLAY-MSG
        PERFORM CLEANUP
           THRU CLEANUP-EXIT
     END-IF.
 APPC-SEND-EXIT. EXIT.
*****************************************************************
*
 APPC-RECEIVE.
*************************************************
* PERFORM THIS CALL UNTIL A "NOT" CM-OK         *
* RETURN CODE IS RECEIVED.  ALLOWING RECEPTION OF: *
* - PROCESSING-RESULTS-RECORD FROM CREDRPT PROGRAM *
* - CONVERSATION DEALLOCATION RETURN CODE       *
*     FROM THE CREDRPT PROGRAM                  *
*************************************************
     MOVE 25 TO REQUESTED-LENGTH.
     CALL "CMRCV" USING CONVERSATION-ID
                        BUFFER
                        REQUESTED-LENGTH
                        DATA-RECEIVED
                        RECEIVED-LENGTH
                        STATUS-RECEIVED
                        REQUEST-TO-SEND-RECEIVED
                        CM-RETCODE.
*
     IF CM-COMPLETE-DATA-RECEIVED
        MOVE BUFFER TO PROCESSING-RESULTS-RECORD
        DISPLAY PROCESSING-RESULTS-RECORD
     END-IF.
```

```
     IF CM-OK OR CM-DEALLOCATED-NORMAL
        NEXT SENTENCE
     ELSE
        MOVE "RECEIVE PROCESSING TERMINATED"
             TO CM-ERROR-DISPLAY-MSG
     END-IF.
 APPC-RECEIVE-EXIT. EXIT.
*****************************************************************
*
 CLEANUP.
*********************************************
* DISPLAY EXECUTION COMPLETE OR ERROR MESSAGE *
* NOTE: CREDRPT WILL DEALLOCATE CONVERSATION  *
*********************************************
     IF CM-ERROR-DISPLAY-MSG  = SPACES
         DISPLAY "PROGRAM: SALESRPT EXECUTION COMPLETE"
     ELSE
         DISPLAY "SALESRPT PROGRAM - ",
                 CM-ERROR-DISPLAY-MSG, " RC= ", CM-RETCODE.
     STOP RUN.
 CLEANUP-EXIT. EXIT.
*****************************************************************
```

CREDRPT (Acceptor of the Conversation)

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.        CREDRPT.
*************************************************************
* THIS IS THE CREDRPT PROGRAM THAT RECIEVES DATA FROM THE   *
* SALESRPT PROGRAM FOR CREDIT BALANCE PROCESSING.           *
*                                                           *
* PURPOSE: RECEIVE A SALES-RECORD FROM THE SALESRPT PROGRAM *
*          AND COMPUTE AND DISPLAY A NEW CREDIT BALANCE,    *
*          THEN SEND A STATUS INDICATION TO SALESRPT.       *
*                                                           *
* INPUT:   SALES-RECORD FROM SALESRPT PROGRAM.              *
*                                                           *
* OUTPUT:  DISPLAY OUTPUT-RECORD.                           *
*          PROCESSING-RESULTS-RECORD TO SALESRPT.           *
*                                                           *
* NOTE:    SALES-RECORD PROCESSING HAS BEEN GREATLY         *
*          SIMPLIFIED IN THIS EXAMPLE.                      *
*************************************************************
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. IBM-370.
 OBJECT-COMPUTER. IBM-370.
 SPECIAL-NAMES.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
 I-O-CONTROL.
*
 DATA DIVISION.
 FILE SECTION.
 WORKING-STORAGE SECTION.

 01  CM-ERROR-DISPLAY-MSG       PIC X(40)  VALUE SPACES.

 01  BUFFER                     PIC X(52).

 01  CURRENT-CREDIT-BALANCE     PIC S9(7)V99.

 01  CONVERSATION-STATUS    PIC 9(9)    COMP-4.
     88 CONVERSATION-ACCEPTED           VALUE 1.
     88 CONVERSATION-NOT-ESTABLISHED    VALUE 0.

***************
* SALES-RECORD *
***************
 01  SALES-RECORD.
     05  CUST-NUM               PIC X(4).
     05  CUST-NAME              PIC X(20).
     05  FILLER                 PIC X(5).
     05  CREDIT-BALANCE         PIC S9(7)V99.
     05  CREDIT-LIMIT           PIC S9(7)V99.
     05  CREDIT-FLAG            PIC X.

****************
* OUTPUT-RECORD *
****************
```

```
 01  OUTPUT-RECORD.
     05 FILLER                   PIC X.
     05 OP-CUST-NUM              PIC X(4).
     05 FILLER                   PIC X(3)   VALUE SPACES.
     05 OP-CUST-NAME             PIC X(20).
     05 FILLER                   PIC X(5)   VALUE SPACES.
     05 OP-CREDIT-LIMIT          PIC Z(6)9.99-.
     05 FILLER                   PIC X(5)   VALUE SPACES.
     05 OP-CREDIT-BALANCE        PIC Z(6)9.99-.
     05 FILLER                   PIC X(5)   VALUE SPACES.
     05 OP-TEXT-FIELD            PIC X(25).
     05 FILLER                   PIC X(5)   VALUE SPACES.

*****************************
* PROCESSING-RESULTS-RECORD *
*****************************
 01  PROCESSING-RESULTS-RECORD   PIC X(25)  VALUE SPACES.

*******************************************
* CPI-COMMUNICATIONS PSEUDONYM COPYBOOK FILE *
*******************************************
     COPY CMCOBOL.

 LINKAGE SECTION.

     EJECT.
*
 PROCEDURE DIVISION.
*********************************************************************
*************************  START OF MAINLINE  ***********************
*********************************************************************
 MAINLINE.

     PERFORM APPC-ACCEPT
        THRU APPC-ACCEPT-EXIT.
     DISPLAY "CREDRPT CONVERSATION ACCEPTED".

     PERFORM APPC-RECEIVE
        THRU APPC-RECEIVE-EXIT
        UNTIL CM-SEND-RECEIVED.
     DISPLAY "CREDRPT RECORD RECEIVED".

     PERFORM PROCESS-RECORD
        THRU PROCESS-RECORD-EXIT.
     DISPLAY "CREDRPT DATA PROCESSED".

     PERFORM APPC-SEND
        THRU APPC-SEND-EXIT.
     DISPLAY "CREDRPT RESULTS RECORD SENT".

     PERFORM CLEANUP
        THRU CLEANUP-EXIT.
     STOP RUN.
*********************************************************************
*************************  END OF MAINLINE  *************************
*********************************************************************
*
 APPC-ACCEPT.
```

```
*************************************************
* ACCEPT INCOMING APPC CONVERSATION ESTABLISHING *
* DEFAULT CONVERSATION CHARACTERISTICS           *
*************************************************
     CALL "CMACCP" USING CONVERSATION-ID
                         CM-RETCODE.
     IF CM-OK
        SET CONVERSATION-ACCEPTED TO TRUE
     ELSE
        MOVE "ACCEPT PROCESSING TERMINATED"
             TO CM-ERROR-DISPLAY-MSG
        PERFORM CLEANUP
           THRU CLEANUP-EXIT
     END-IF.
 APPC-ACCEPT-EXIT. EXIT.
*****************************************************************
*
 APPC-RECEIVE.
*************************************************************
* PERFORM THIS CALL UNTIL A CM-SEND-RECEIVE INDICATION IS   *
* RECEIVED. THIS INDICATES A CONVERSATION STATE CHANGE FROM *
* RECEIVE TO SEND OR SEND-PENDING STATE, THUS "CMRCV"       *
* (RECEIVE) HAS COMPLETED. ALLOWING RECEPTION OF:           *
* - SALES-RECORD FROM SALESRPT PROGRAM                      *
*************************************************************
     MOVE 52 TO REQUESTED-LENGTH.
     CALL "CMRCV" USING CONVERSATION-ID
                        BUFFER
                        REQUESTED-LENGTH
                        DATA-RECEIVED
                        RECEIVED-LENGTH
                        STATUS-RECEIVED
                        REQUEST-TO-SEND-RECEIVED
                        CM-RETCODE.
*
     IF CM-COMPLETE-DATA-RECEIVED
        MOVE BUFFER TO SALES-RECORD
     END-IF.
*
     IF CM-OK
        NEXT SENTENCE
     ELSE
        PERFORM APPC-SET-DEALLOCATE-TYPE
           THRU APPC-SET-DEALLOCATE-TYPE-EXIT
        MOVE "RECEIVE PROCESSING TERMINATED"
             TO CM-ERROR-DISPLAY-MSG
        PERFORM CLEANUP
           THRU CLEANUP-EXIT
     END-IF.
 APPC-RECEIVE-EXIT. EXIT.
*****************************************************************
*
 PROCESS-RECORD.
     SUBTRACT CREDIT-BALANCE FROM CREDIT-LIMIT
        GIVING CURRENT-CREDIT-BALANCE.
     IF CREDIT-FLAG = "0"
        MOVE "**CREDIT LIMIT EXCEEDED**" TO OP-TEXT-FIELD
     ELSE
```

```
           MOVE SPACES TO OP-TEXT-FIELD
        END-IF.
        MOVE CUST-NUM TO OP-CUST-NUM.
        MOVE CUST-NAME TO OP-CUST-NAME.
        MOVE CREDIT-LIMIT TO OP-CREDIT-LIMIT.
        MOVE CURRENT-CREDIT-BALANCE TO OP-CREDIT-BALANCE.
        DISPLAY OUTPUT-RECORD.
*
        MOVE "CREDIT RECORD UPDATED" TO PROCESSING-RESULTS-RECORD.
 PROCESS-RECORD-EXIT. EXIT.
*****************************************************************
*
 APPC-SEND.
        MOVE PROCESSING-RESULTS-RECORD TO BUFFER.
        MOVE 25 TO SEND-LENGTH.

************************************************
* SEND THE PROCESSING-RESULTS-RECORD TO SALESRPT *
************************************************
        CALL "CMSEND" USING CONVERSATION-ID
                            BUFFER
                            SEND-LENGTH
                            REQUEST-TO-SEND-RECEIVED
                            CM-RETCODE.
        IF CM-OK
           NEXT SENTENCE
        ELSE
           PERFORM APPC-SET-DEALLOCATE-TYPE
              THRU APPC-SET-DEALLOCATE-TYPE-EXIT
           MOVE "SEND PROCESSING TERMINATED"
                TO CM-ERROR-DISPLAY-MSG
           PERFORM CLEANUP
              THRU CLEANUP-EXIT
        END-IF.
 APPC-SEND-EXIT. EXIT.
*****************************************************************
*
 APPC-SET-DEALLOCATE-TYPE.
        SET CM-DEALLOCATE-ABEND TO TRUE.

***************************************
* ON ERROR SET DEALLOCATE-TYPE TO ABEND *
***************************************
        CALL "CMSDT" USING CONVERSATION-ID
                           DEALLOCATE-TYPE
                           CM-RETCODE.
        IF CM-OK
           NEXT SENTENCE
        ELSE
           DISPLAY "ERROR SETTING CONVERSATION DEALLOCATE TYPE"
        END-IF.
 APPC-SET-DEALLOCATE-TYPE-EXIT. EXIT.
*****************************************************************
*
 CLEANUP.
        IF CONVERSATION-ACCEPTED
*******************************
* DEALLOCATE APPC CONVERSATION *
```

```
*******************************
        CALL "CMDEAL" USING CONVERSATION-ID
                            CM-RETCODE
        DISPLAY "CREDRPT DEALLOCATED CONVERSATION"
    END-IF.
    IF CM-ERROR-DISPLAY-MSG  = SPACES
        DISPLAY "PROGRAM: CREDRPT EXECUTION COMPLETE"
    ELSE
        DISPLAY "CREDRPT PROGRAM - ",
                CM-ERROR-DISPLAY-MSG, " RC= ", CM-RETCODE
    END-IF.
    STOP RUN.
CLEANUP-EXIT. EXIT.
*****************************************************************
```

Pseudonym File for COBOL

The source shown below is found in "CMCOBOL COPY" and defines the pseudonyms that are used in the sample programs. See "Programming Language Considerations" on page 177 and Appendix A, "Variables and Characteristics" for further discussion on the values of conversation characteristics.

```
*COPY CMCOBOL
      ************************************************
      * NOTE: BUFFER MUST BE DEFINED IN WORKING STORAGE *
      ************************************************
      *
       01  CONVERSATION-ID           PIC X(8).
      *
       01  CONVERSATION-TYPE         PIC 9(9) COMP-4.
           88  CM-BASIC-CONVERSATION   VALUE 0.
           88  CM-MAPPED-CONVERSATION  VALUE 1.
      *
       01  CM-RETCODE                    PIC 9(9) COMP-4.
      *    ===> RETURN-CODE IS A RESERVED WORD IN SOME  <===
      *    ===> VERSIONS OF COBOL                       <===
      *
           88   CM-OK                           VALUE 0.
           88   CM-ALLOCATE-FAILURE-NO-RETRY    VALUE 1.
           88   CM-ALLOCATE-FAILURE-RETRY       VALUE 2.
           88   CM-CONVERSATION-TYPE-MISMATCH   VALUE 3.
           88   CM-SECURITY-NOT-VALID           VALUE 6.
           88   CM-SYNC-LVL-NOT-SUPPORTED-PGM   VALUE 8.
           88   CM-TPN-NOT-RECOGNIZED           VALUE 9.
           88   CM-TP-NOT-AVAILABLE-NO-RETRY    VALUE 10.
           88   CM-TP-NOT-AVAILABLE-RETRY       VALUE 11.
           88   CM-DEALLOCATED-ABEND            VALUE 17.
           88   CM-DEALLOCATED-NORMAL           VALUE 18.
           88   CM-PARAMETER-ERROR              VALUE 19.
           88   CM-PRODUCT-SPECIFIC-ERROR       VALUE 20.
           88   CM-PROGRAM-ERROR-NO-TRUNC       VALUE 21.
           88   CM-PROGRAM-ERROR-PURGING        VALUE 22.
           88   CM-PROGRAM-ERROR-TRUNC          VALUE 23.
           88   CM-PROGRAM-PARAMETER-CHECK      VALUE 24.
           88   CM-PROGRAM-STATE-CHECK          VALUE 25.
           88   CM-RESOURCE-FAILURE-NO-RETRY    VALUE 26.
           88   CM-RESOURCE-FAILURE-RETRY       VALUE 27.
           88   CM-UNSUCCESSFUL                 VALUE 28.
      *
       01  DATA-RECEIVED             PIC 9(9) COMP-4.
           88  CM-NO-DATA-RECEIVED         VALUE 0.
           88  CM-DATA-RECEIVED            VALUE 1.
           88  CM-COMPLETE-DATA-RECEIVED   VALUE 2.
           88  CM-INCOMPLETE-DATA-RECEIVED VALUE 3.
      *
       01  DEALLOCATE-TYPE           PIC 9(9) COMP-4.
           88   CM-DEALLOCATE-SYNC-LEVEL   VALUE 0.
           88   CM-DEALLOCATE-FLUSH        VALUE 1.
           88   CM-DEALLOCATE-CONFIRM      VALUE 2.
           88   CM-DEALLOCATE-ABEND        VALUE 3.
      *
       01  ERROR-DIRECTION           PIC 9(9) COMP-4.
           88   CM-RECEIVE-ERROR     VALUE 0.
```

```
     88   CM-SEND-ERROR          VALUE 1.
*
 01  FILL                        PIC 9(9) COMP-4.
     88   CM-FILL-LL             VALUE 0.
     88   CM-FILL-BUFFER         VALUE 1.
*
 01  LOG-DATA                    PIC X(512).
*    0-512 BYTES
*
 01  LOG-DATA-LENGTH             PIC 9(9) COMP-4.
*
 01  MODE-NAME                   PIC X(8).
*    0-8 BYTES
*
 01  MODE-NAME-LENGTH            PIC 9(9) COMP-4.
*
 01  PARTNER-LU-NAME             PIC X(17).
*    1-17 BYTES
*
 01  PARTNER-LU-NAME-LENGTH      PIC 9(9) COMP-4.
*
 01  PREPARE-TO-RECEIVE-TYPE     PIC 9(9) COMP-4.
     88   CM-PREP-TO-RECEIVE-SYNC-LEVEL  VALUE 0.
     88   CM-PREP-TO-RECEIVE-FLUSH       VALUE 1.
     88   CM-PREP-TO-RECEIVE-CONFIRM     VALUE 2.
*
 01  RECEIVED-LENGTH             PIC 9(9) COMP-4.
*
 01  RECEIVE-TYPE                PIC 9(9) COMP-4.
     88   CM-RECEIVE-AND-WAIT    VALUE 0.
     88   CM-RECEIVE-IMMEDIATE   VALUE 1.
*
 01  REQUESTED-LENGTH            PIC 9(9) COMP-4.
*
 01  REQUEST-TO-SEND-RECEIVED    PIC 9(9) COMP-4.
     88  CM-REQ-TO-SEND-NOT-RECEIVED    VALUE 0.
     88  CM-REQ-TO-SEND-RECEIVED        VALUE 1.
*
 01  RETURN-CONTROL              PIC 9(9) COMP-4.
     88   CM-WHEN-SESSION-ALLOCATED      VALUE 0.
     88   CM-IMMEDIATE                   VALUE 1.
*
 01  SEND-LENGTH                 PIC 9(9) COMP-4.
*
 01  SEND-TYPE                   PIC 9(9) COMP-4.
     88   CM-BUFFER-DATA                 VALUE 0.
     88   CM-SEND-AND-FLUSH              VALUE 1.
     88   CM-SEND-AND-CONFIRM            VALUE 2.
     88   CM-SEND-AND-PREP-TO-RECEIVE    VALUE 3.
     88   CM-SEND-AND-DEALLOCATE         VALUE 4.
*
 01  STATUS-RECEIVED             PIC 9(9) COMP-4.
     88  CM-NO-STATUS-RECEIVED        VALUE 0.
     88  CM-SEND-RECEIVED             VALUE 1.
     88  CM-CONFIRM-RECEIVED          VALUE 2.
     88  CM-CONFIRM-SEND-RECEIVED     VALUE 3.
     88  CM-CONFIRM-DEALLOC-RECEIVED  VALUE 4.
*
```

```
 01  SYNC-LEVEL                PIC 9(9) COMP-4.
     88   CM-NONE               VALUE 0.
     88   CM-CONFIRM            VALUE 1.
*
 01  SYM-DEST-NAME             PIC X(8).
*
 01  TP-NAME                   PIC X(64).
*    1-64 BYTES
*
 01  TP-NAME-LENGTH            PIC 9(9) COMP-4.
```

Results of Successful Program Execution

SALESRPT program:

```
SALESRPT CONVERSATION INTIALIZED
SALESRPT CONVERSATION ALLOCATED
SALESRPT DATA RECORD SENT
SALESRPT RESULTS RECORD RECEIVED
PROGRAM: SALESRPT EXECUTION COMPLETE
```

CREDRPT Program:

```
CREDRPT CONVERSATION ACCEPTED
CREDRPT RECORD RECEIVED
 0010   XYZ INC.                    5000.00          724.50
CREDRPT DATA PROCESSED
CREDRPT RESULTS RECORD SENT
CREDRPT DEALLOCATED CONVERSATION
PROGRAM: CREDRPT EXECUTION COMPLETE
```

Glossary

B

basic conversation. A conversation in which programs exchange data records in an SNA-defined format. This format is a stream of data containing 2-byte length prefixes that specify the amount of data to follow before the next prefix.

C

conversation. A logical connection between two programs over an LU type 6.2 session that allows them to communicate with each other while processing a transaction. See also basic conversation and mapped conversation.

conversation characteristics. The attributes of a conversation that determine the functions and capabilities of programs within the conversation.

conversation partner. One of the two programs involved in a conversation.

conversation state. The condition of a conversation that reflects what the past action on that conversation has been and that determines what the next set of actions may be.

Common Programming Interface. Provides languages, commands, and calls that allow the development of applications that are more easily integrated and moved across environments supported by Systems Applications Architecture.

L

local program. The program being discussed within a particular context. Contrast with remote program.

logical unit. A port providing formatting, state synchronization, and other high-level services through which an end user communicates with another end user over an SNA network.

logical unit type 6.2. The SNA logical unit type that supports general communication between programs in a distributed processing environment; the SNA logical unit type on which CPI Communications is built.

M

mapped conversation. A conversation in which programs exchange data records with arbitrary data formats agreed upon by the applications programmers.

P

partner. See conversation partner.

privilege. An identification that a product or installation defines in order to differentiate SNA service transaction programs from other programs, such as application programs.

R

remote program. The program at the other end of a conversation with respect to the reference program. Contrast with local program.

S

session. A logical connection between two logical units that can be activated, tailored to provide various protocols, and deactivated as requested.

side information. System-defined values that are used for the initial values of the *partner_LU_name*, *mode_name*, and *TP_name* characteristics.

state. See conversation state.

state transition. The act of moving from one conversation state to another.

symbolic destination name. Variable corresponding to an entry in the side information.

Systems Application Architecture. A set of software interfaces, conventions, and protocols that provide a framework for designing and developing applications with cross-system consistency.

Systems Network Architecture. A description of the logical structure, formats, protocols, and operational sequences for transmitting information units through, and controlling the configuration and operation of, networks.

T

transition. See state transition.

Index

D

E

F

G

H

I

S

T

U

V

W

X

Special Characters

Systems Application Architecture
Common Programming Interface
Communications Reference
(SC26-4399-1)

Reader's Comment Form

Use this form to send in your comments about this book, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be forwarded to the author for review and appropriate action.

Note: Do not use this form to request IBM publications. If you do, your order will be delayed because publications are not stocked at the address printed on the reverse side. Instead, you should direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to your IBM branch office.

☐ Please check here if you wish a reply.

Name _____ Phone ( ) _____

Company _____ Job Title _____

Address _____

_____

Number of latest TNL applied to this publication: _____

Comments:

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM representative will be happy to forward your comments or you may mail them directly to the address in the Edition Notice on the back of the title page.)

SC26-4399-1

Reader's Comment Form

Fold and tape | Please Do Not Staple | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Dept. E96
P.O. Box 12195
Research Triangle Park, N.C. 27709-9990

Fold and tape | Please Do Not Staple | Fold and tape

IBM®

File Number
S370-40

Systems Application Architecture Library

An Overview	GC26-4341
Common User Access: Panel Design and User Interaction	SC26-4351
Writing Applications: A Design Guide	SC26-4362
Common Programming Interface:	
Application Generator Reference	SC26-4355
C Reference	SC26-4353
COBOL Reference	SC26-4354
Communications Reference	SC26-4399
Database Reference	SC26-4348
Dialog Reference	SC26-4356
FORTRAN Reference	SC26-4357
Presentation Reference	SC26-4359
Procedures Language Reference	SC26-4358
Query Reference	SC26-4349

SC26-4399-1

Printed in U.S.A.