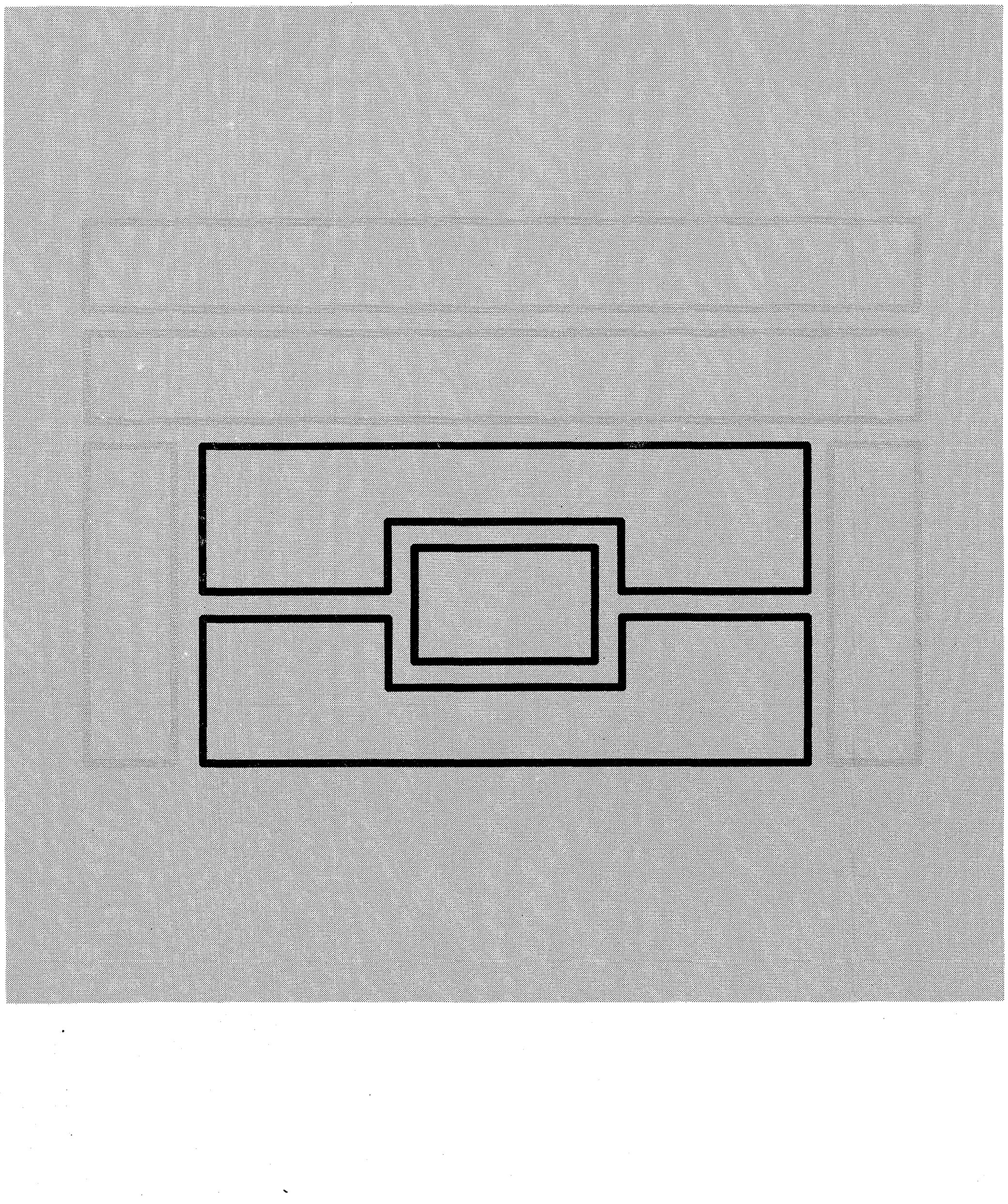
Systems Application Architecture

SC26-4399-03

Common Programming Interface Communications Reference



Systems Application Architecture

SC26-4399-03

Common Programming Interface Communications Reference

Fourth Edition (March 1991)

This edition replaces and makes obsolete the previous edition, SC26-4399-2.

This edition applies to the IBM* Systems Application Architecture* Common Programming Interface Communications and to the following:

- Version 3 Release 2 Modification 1 of CICS/ESA (program number 6585-083)
- Version 3 Release 2 of IMS/ESA (program number 5665-409)
- Version 4 Release 2 of MVS/ESA System Product (program numbers 5695-047 and 5695-048)
- Version 1.0 of Networking Services/2 (program number 86F2503)
- Version 2 Release 1 Modification 0 of Operating System/400 (program number 5738-SS1)
- Release 1.0 of VM/Enterprise Systems Architecture (program number 5684-112)

and to all subsequent releases and modifications until otherwise indicated in new editions. Consult the latest edition of the applicable IBM system bibliography for current information on these products.

Specific changes are indicated by a vertical bar to the left of the change. Editorial changes that have no technical significance are not noted. For a detailed list of changes, see "Summary of Changes" on page 375.

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

A form for readers' comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation
Dept. F14
P.O. Box 12195
Research Triangle Park, NC, U.S.A. 27709-9990.

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

Notices

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

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

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.

This *SAA Common Programming Interface: Communications Reference* is intended to help the customer program with CPI Communications. This *SAA Common Programming Interface: Communications Reference* documents General-Use Programming Interface and Associated Guidance Information provided by CPI Communications as implemented in CICS/ESA, IMS/ESA, MVS/ESA, Networking Services/2, OS/400, and VM/ESA.

General-Use programming interfaces allow the customer to write programs that obtain the services of CPI Communications as implemented in CICS/ESA, IMS/ESA, MVS/ESA, Networking Services/2, OS/400, and VM/ESA.

The following terms, denoted by an asterisk (*) on their first occurrence in this publication, are trademarks or service marks of the IBM Corporation in the United States and/or other countries:

AD/Cycle	
AS/400	Application System/400
CICS/ESA	
CMS	Conversational Monitor System
ESA/370	Enterprise Systems Architecture/370
ESA/390	Enterprise Systems Architecture/390
IBM	
IMS/ESA	
MVS/ESA	
OS/2	Operating System/2
OS/400	Operating System/400
SAA	Systems Application Architecture
VM/ESA	Virtual Machine/Enterprise Systems Architecture

Contents

Figures

Tables

Chapter 1. Introduction

This introductory chapter:

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

Who Should Read This Book

This book defines the Communications element of SAA's Common Programming Interface (CPI). CPI Communications, also known as the SAA communications interface, provides a programming interface that allows program-to-program communications using IBM's Systems Network Architecture (SNA) logical unit 6.2 (LU 6.2). This book is intended for programmers who want to write applications that adhere to this definition.

Although this 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 the SAA Solution?

The SAA solution is based on a set of software interfaces, conventions, and protocols that provide a framework for designing and developing applications.

The SAA solution:

- Defines a **common programming interface** that can be used to develop consistent, integrated enterprise software
- Defines **common communications support** for the connection of applications, systems, networks, and devices
- Defines a **common user access** that allows consistency in screen layout and user interaction techniques
- Offers some **applications** and **application development tools** written by IBM.

Supported Environments

Several combinations of IBM* hardware and software have been selected as SAA environments. These are environments in which IBM will manage the availability of support for applicable SAA elements, and the conformance of those elements to SAA specifications. The SAA environments are the following:

- MVS
  - Base system (TSO/E, APPC/MVS, batch)
  - CICS
  - IMS
- VM with CMS
- Operating System/400* (OS/400*)
- Operating System/2* (OS/2*).

Common Programming Interface

The common programming interface (CPI) provides languages and services that can be used to develop applications which take advantage of SAA consistency. 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
  PL/I
  Procedures Language
  RPG

- Services

  Communications Interface (defined by this book)
  Database Interface
  Dialog Interface
  Presentation Interface
  PrintManager Interface
  Query Interface
  Repository Interface
  Resource Recovery Interface.

The CPI is defined by this book and the other CPI reference books. The CPI 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, and it establishes a common base across the applicable SAA environments.

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 comprehensive understanding of SAA CPI Communications.

How This Book Is Organized

In addition to this introductory chapter, the 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 use CPI Communications calls for program-to-program communication.

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

  This chapter describes the 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, "CPI Communications on CICS/ESA"**

  This appendix contains information about the CICS/ESA implementation of CPI Communications.

- **Appendix F, "CPI Communications on IMS/ESA"**

  This appendix contains information about the IMS/ESA implementation of CPI Communications.

- **Appendix G, "CPI Communications on MVS/ESA"**

  This appendix contains information about the MVS/ESA implementation of CPI Communications.

- **Appendix H, "CPI Communications on OS/2"**

  This appendix contains information about the OS/2 implementation of CPI Communications, including OS/2 extensions to CPI Communications.

- **Appendix I, "CPI Communications on Operating System/400"**

  This appendix contains information about the OS/400 implementation of CPI Communications.

- **Appendix J, "CPI Communications on VM/ESA CMS"**

  This appendix contains information about the VM/ESA* implementation of CPI Communications, including VM/ESA extensions to CPI Communications.

- **Appendix K, "Sample Programs"**

  This appendix contains two sample COBOL programs using CPI Communications.

- **Appendix L, "Pseudonym Files"**

  This appendix contains CPI Communications pseudonym files for all of the SAA languages.

General Path through the Manual

This book contains both tutorial and reference information. Use the following path 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."
- Read Chapter 3, "Program-to-Program Communication Tutorial" for explanation and examples of how to use the CPI Communications calls. When reading this chapter, use Chapter 4, "Call Reference Section" to obtain additional information about the function of and required parameters for the CPI Communications calls.
- Use Chapter 4, "Call Reference Section" and the appendixes for specific functional information on how to code applications.

Relationship to Products

The SAA CPI Communications interface defines elements that are consistent across the applicable SAA environments. Preparing and running programs requires the use of a CPI Communications product that implements the interface on one of these systems. For the CPI Communications interface, the implementing products are:

- CICS/ESA
- IMS/ESA
- MVS/ESA
- Networking Services/2 (on OS/2)
- OS/400
- VM.

These products have their own sets of product documentation, which are required in addition to this one. This book defines the interface elements that are common across the environments. The product appendixes in this book and other product documents describe any additional elements and—more importantly—explain how to prepare and run a program in a particular environment.

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

How Product Implementations Are Designated

Because the SAA solution is still evolving, complete and consistent products may not be available yet on all applicable 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 or announced 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 CPI Communications, all of the interface elements are implemented or announced for the applicable systems:

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	X

Note: In the interface definition table for each CPI Communications call, the MVS column refers to MVS/ESA with or without TSO/E. The OS/2 column refers to Networking Services/2 on OS/2. The VM column refers to VM/ESA with CMS. For information about using CPI Communications with VM/SP, see the second edition of this book (order number ST00-3945).

- The SAA Communications interface definition is printed in black ink. If the implementation of an interface element in an operating environment differs from the SAA definition 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 22.

Related Publications

Additional guidance on program-to-program communications is provided in *Program-To-Program Communications in Systems Application Architecture (SAA) Environments*, GG24-3482. This book describes the VM implementation of CPI Communications as well as prototype CPI Communications implementations for OS/2 and OS/400. The prototypes do not represent actual product implementations of CPI Communications. This book contains no programming interfaces for customer use.

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

For the SAA Solution

An introduction to the SAA solution in general can be found in *SAA: An Overview*, GC26-4341.

An introduction to the common programming interface can be found in *Common Programming Interface: Summary*, GC26-4675.

More detailed information on the components of the common programming interface is available in the following SAA manuals (including this one):

Application Generator Reference, SC26-4355
C Reference—Level 2, SC09-1308
COBOL Reference, SC26-4354
Communications Reference, SC26-4399
Database Reference, SC26-4348
Dialog Reference, SC26-4356
FORTRAN Reference, SC26-4357
PL/I Reference, SC26-4381
Presentation Reference, SC26-4359
Procedures Language Reference, SC26-4358
Procedures Language Level 2 Reference, SC24-5549

Query Reference, SC26-4349
Repository Reference, SC26-4684
Resource Recovery Reference, SC31-6821
RPG Reference, SC09-1286.

General programming advice may be found in *Writing Applications: A Design Guide*, SC26-4362. An introduction to the use of the AD/Cycle* application development tools can be found in *AD/Cycle Concepts*, GC26-4531.

A definition of the common user access can be found in *Common User Access: Advanced Interface Design Guide*, SC26-4582, and *Common User Access: Basic Interface Design Guide*, SC26-4583.

More information on the common communications support can be found in *Common Communications Support: Summary*, GC31-6810.

An introduction to distributed data in the SAA world can be found in *Concepts of Distributed Data*, SC26-4417.

More information on SAA system management can be found in *An Introduction to SystemView*, GC23-0576.

Ordering Information: Contact your local IBM branch office for information on how to order the above publications. They also can be obtained through an authorized IBM dealer. The entire set of SAA publications can be ordered by specifying the bill-of-forms number SBOF-1240.

For Implementing Products

Information about using CPI Communications in each SAA operating environment can be found in the following books.

CICS

- *CICS/ESA Intercommunication Guide*, SC33-0657
- *CICS/ESA Application Programming Guide*, SC33-0675
- *CICS/ESA Application Programmer's Reference*, SC33-0676
- *CICS Library Guide*, GC33-0356

IMS

- *IBM Licensed Program Specifications* (for IMS), GC26-4289
- *IMS/ESA General Information*, GC26-4275
- *IMS/ESA Release Planning Guide*, GC26-4630
- *IMS/ESA Installation Guide*, SC26-4276
- *IMS/ESA System Definition Reference*, SC26-4278
- *IMS/ESA Application Programming: Design Guide*, SC26-4279
- *IMS/ESA Data Communication Administration Guide*, SC26-4286
- *IMS/ESA Application Programming: Data Communication*, SC26-4283
- *IMS/ESA Customization Guide: Data Communication*, SC26-4625
- *IMS/ESA System Administration Guide*, SC26-4282
- *IMS/ESA Utilities Reference: Data Communication*, SC26-4628
- *IMS/ESA Utilities Reference: System*, SC26-4629
- *IMS/ESA Operations Guide*, SC26-4287
- *IMS/ESA Sample Operating Procedures*, SC26-4277
- *IMS/ESA Operator's Reference*, SC26-4288
- *IMS/ESA Summary of Operator Commands*, SX26-3764
- *IMS/ESA Messages and Codes*, SC26-4290

- *IMS/ESA Failure Analysis Structure Tables (FAST) for Dump Analysis*, LY27-9512
- *IMS/ESA Diagnosis Guide and Reference*, LY27-9539
- *IMS/ESA Master Index and Glossary*, SC26-4291
- *LU 6.1 Adapter for LU 6.2 Applications: Program Description/Operations*, SC26-4392

MVS

- *MVS/ESA Application Development: Writing Transaction Programs for APPC/MVS*, GC28-1121
- *MVS/ESA Planning: APPC Management*, GC28-1110
- MVS/ESA System Messages Vols 1-3, GC28-1656, GC28-1657, GC28-1658
- APPC/MVS Handbook for the OS/2 System Administrator, GC28-1133

OS/2

- *IBM SAA Networking Services/2: Installation and Network Administrator's Guide*, SC52-1110
- *IBM SAA Networking Services/2: System Management Reference*, SC52-1111
- *IBM SAA Networking Services/2: APPC Programming Reference*, SC52-1112
- *IBM SAA Networking Services/2: Problem Determination Guide*, SC52-1113

OS/400

- *Application System/400 Communications: Advanced Program-to-Program Communications Programmer's Guide*, SC41-8189
- *Application System/400 Programming: Work Management Guide*, SC41-8078
- *Application System/400 Communications: Management Guide*, SC41-0024
- *Application System/400 Communications: Advanced Peer-to-Peer Networking User's Guide*, SC41-8188
- *Application System/400 Programming: Control Language Reference*, SC41-0030
- *Application System/400 Programming: Procedures Language 400/REXX Programmer's Guide*, SC24-5553

VM

- *VM/ESA CMS Command Reference*, SC24-5461
- *VM/ESA CMS Application Development Reference*, SC24-5451
- *VM/ESA CMS Application Development Reference for Assembler*, SC24-5453
- *VM/ESA CMS Application Development Guide*, SC24-5450
- *VM/ESA CMS Planning and Administration Guide*, SC24-5445
- *VM/ESA CMS Administration Reference*, SC24-5446
- *VM/ESA CP Programming Services for 370*, SC24-5435
- *VM/ESA System Messages and Codes for 370*, SC24-5437
- *VM/ESA Connectivity Planning, Administration, and Operation*, SC24-5448
- *VM/ESA CP Programming Services*, SC24-5520
- *VM/ESA System Messages and Codes Reference*, SC24-5529
- *VM/ESA Procedures Language VM/REXX Reference*, SC24-5465.

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

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.

Revision bars are used for this table only if a new element is added to the interface. Revision bars are not used to show new implementations of an existing element.

Table 1 (Page 1 of 2). Major Elements of the Communications Interface

Call Name	MVS[1]	VM	OS/400	OS/2	IMS	CICS
Starter Set						
Accept_Conversation	X	X	X	X	X	X
Allocate	X	X	X	X	X	X
Deallocate	X	X	X	X	X	X
Initialize_Conversation	X	X	X	X	X	X
Receive	X	X	X	X	X	X
Send_Data	X	X	X	X	X	X
Advanced Function						
for synchronization and control:						
Confirm	X	X	X	X	X	X
Confirmed	X	X	X	X	X	X
Flush	X	X	X	X	X	X
Prepare_To_Receive	X	X	X	X	X	X
Request_To_Send	X	X	X	X	X	X
Send_Error	X	X	X	X	X	X
Test_Request_To_Send_Received	X	X	X	X	X	X
for modifying conversation characteristics:						
Set_Conversation_Type	X	X	X	X	X	X
Set_Deallocate_Type	X	X	X	X	X	X
Set_Error_Direction	X	X	X	X	X	X
Set_Fill	X	X	X	X	X	X
Set_Log_Data	X[2]	X	X	X	X[2]	X
Set_Mode_Name	X	X	X	X	X	X
Set_Partner_LU_Name	X	X	X	X	X	X
Set_Prepare_To_Receive_Type	X	X	X	X	X	X
Set_Receive_Type	X	X	X	X	X	X
Set_Return_Control	X	X	X	X	X	X
Set_Send_Type	X	X	X	X	X	X
Set_Sync_Level						
CM_NONE	X	X	X	X	X	X
CM_CONFIRM	X	X	X	X	X	X
CM_SYNC_POINT		X				
Set_TP_Name	X	X	X	X	X	X

Table 1 (Page 2 of 2). Major Elements of the Communications Interface

Call Name	MVS[1]	VM	OS/400	OS/2	IMS	CICS
for examining conversation characteristics:						
Extract_Conversation_State		X		X		
Extract_Conversation_Type	X	X	X	X	X	X
Extract_Mode_Name	X	X	X	X	X	X
Extract_Partner_LU_Name	X	X	X	X	X	X
Extract_Sync_Level	X	X	X	X	X	X

Notes:

[1] When the MVS box is checked in this table and in the individual call descriptions in Chapter 4, Call Reference Section, it means the call is available under MVS/ESA with or without TSO/E.

[2] For IMS and MVS, this call does update the *log_data* conversation chai However, the error information is not logged or sent to the conversation situations.

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 interprogram 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.

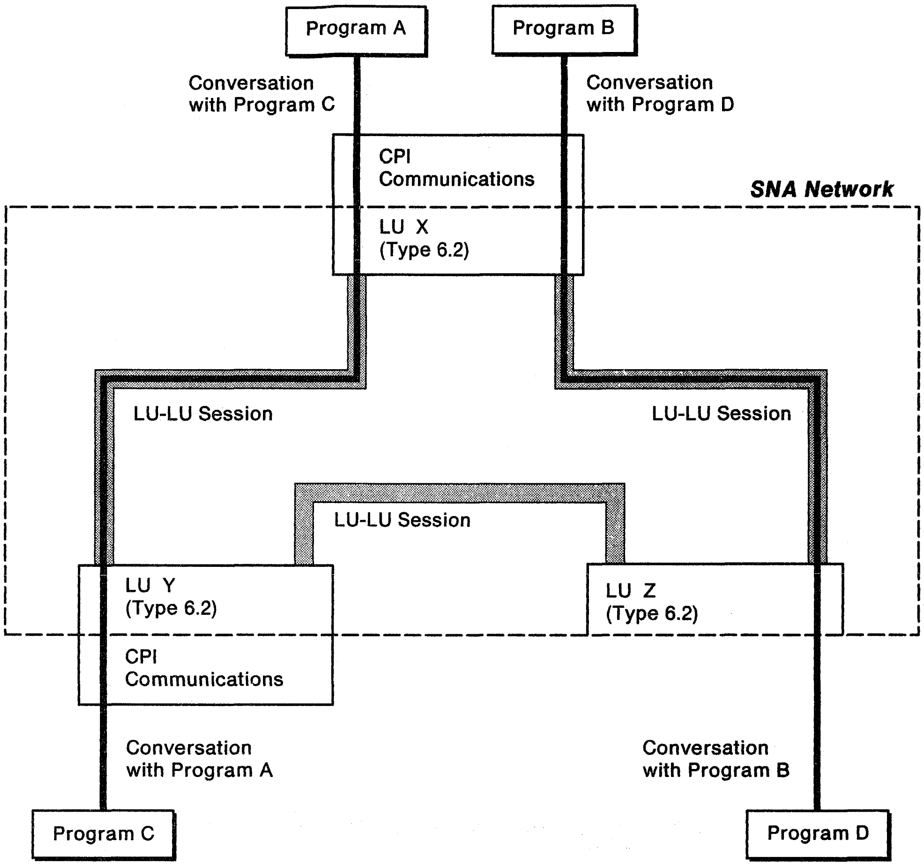


Figure 1. Programs Using CPI Communications to Converse Through an SNA Network

The LUs and the sessions shown in Figure 1 are of type 6.2. Although SNA defines many types of LUs, CPI Communications uses only 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 is intended for programmers using advanced functions. A more complete discussion of basic and mapped conversations is provided in the "Usage Notes" section of "Send_Data (CMSEND)" on page 107.

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.

Although program partners are generally thought of as residing in different nodes in a network, the local and remote programs may, in fact, reside in the same node. Two CPI Communications programs communicate with each other the same way, whether they are in the same or different nodes.

CICS application programs on the same host can communicate using CPI Communications if they are running on different CICS systems, but not if they are running on the same system. The ability for CICS applications to communicate with other CICS applications executing on the same CICS system is provided by other CICS services that do not involve communications protocols. Multiple CICS systems can run on a single host, using VTAM-supported LU 6.2 inter-system communication.

Note: A CPI Communications program may establish a conversation with a program that is using the LU 6.2 application programming interface for a particular environment and not CPI Communications. The conversation between Program B and Program D in Figure 1 is an example of a CPI Communications program communicating with a program that is using a product-specific LU 6.2 application programming interface. Some restrictions may apply in this situation, since CPI Communications does not support all LU 6.2 functions. See Appendix D, "CPI Communications and LU 6.2" on page 181 for a more complete discussion.

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, two lines are 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 12. 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."

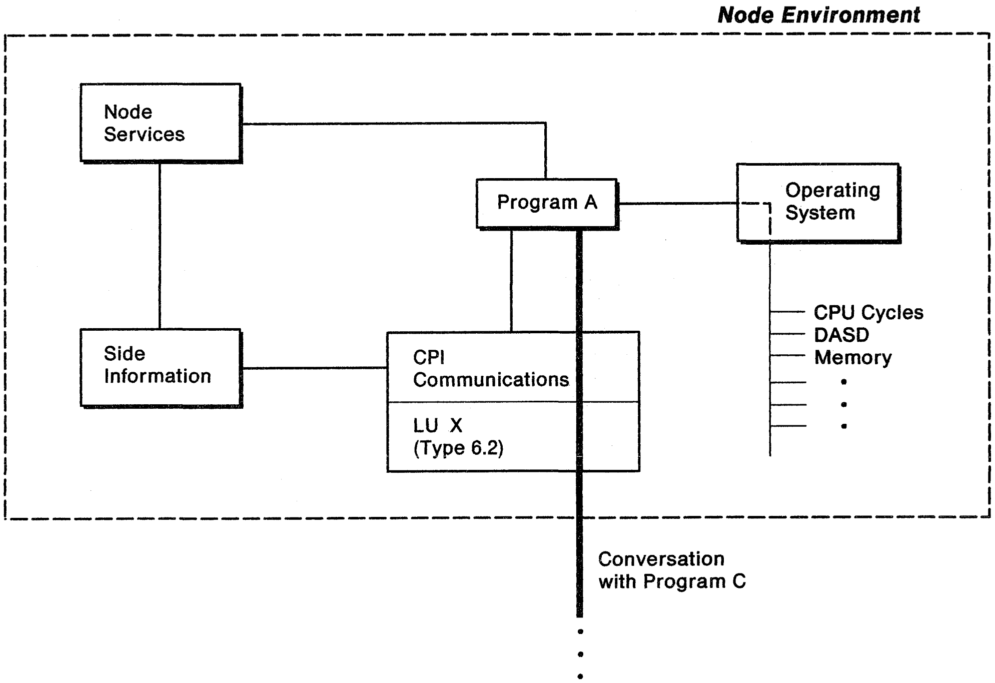


Figure 2. Operating Environment of CPI Communications Program

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 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 used by the local LU to establish sessions with its partners.

- ***TP_name***

  Specifies the name of the remote program. *TP_name* stands for "transaction program name," which comes from the LU 6.2 architecture, where 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. See Appendix D, "CPI Communications and LU 6.2" on page 181 for details of how a CPI Communications program can interact with non-CPI Communications programs.

Some advanced CPI Communications programs may elect to supply their own destination information and not use side information values. Such programs are less dependent on side information definitions and, therefore, can be more flexible. In this case, *sym_dest_name* is left blank. For more information, see "Initialize_Conversation (CMINIT)" on page 90.

IMS and MVS do not support blank *sym_dest_name* values on the Initialize_Conversation call.

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 18. (Refer to specific product information for details.)

- Program-startup 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 32) 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 will abnormally deallocate any dangling conversations. See the appropriate product appendix in this book for a description of how dangling conversations are deallocated in a specific environment.

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 and to exchange data and control information between the programs. (Conversation characteristics are discussed in greater detail in the next section, "Conversation Characteristics.")

When a program makes a CPI Communications call, the program passes characteristics and data to CPI Communications using **input parameters.** When the call completes, CPI Communications passes data and status information back to the program using **output parameters.**

The *return_code* output parameter is returned for all CPI Communications calls. It indicates whether a call completed successfully or if some error was detected that caused the call to fail. CPI Communications uses additional output parameters on some calls to pass status information to the program. These parameters include the *request_to_send_received* parameter, the *data_received* parameter, and the *status_received* parameter.

The function provided by 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 are provided in "Starter-Set Flows" on page 32.

- **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 are provided in "Advanced-Function Flows" on page 39.

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, are often used together with advanced-function calls. The distinction between the two types of calls is intended to aid the CPI Communications programmer and to indicate the relative degree of complexity.

Table 2 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
Extract_Conversation_State	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, "Call Reference Section" on page 63.

Conversation Characteristics

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 prepare 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 startup 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.

Note: In some implementing environments, Program A can share the conversation_ID with another task, allowing that task to also issue calls on the conversation with Program C. See the appropriate product appendix for information about the scope of the conversation_ID in a particular environment.

After issuing their respective Initialize_Conversation and Accept_Conversation calls, both Program A and Program 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 90) and the Accept_Conversation call (described on page 65). The values shown in the figure are pseudonyms 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 22.

Table 3. Characteristics and Their Default Values

Name of Characteristic	Initialize_Conversation sets it to:	Accept_Conversation sets it to:
conversation_state	CM_INITIALIZE_STATE	CM_RECEIVE_STATE
conversation_type	CM_MAPPED_CONVERSATION	The value received on the conversation startup 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*. If a blank *sym_dest_name* was specified, *mode_name* will be the null string.	The mode name for the session on which the conversation startup request arrived
mode_name_length	The length of *mode_name*. If a blank *sym_dest_name* was specified, *mode_name_length* will be 0.	The length of *mode_name*
partner_LU_name	The partner LU name from side information referenced by *sym_dest_name*. If a blank *sym_dest_name* was specified, *partner_LU_name* will be a single blank.	The partner LU name for the session on which the conversation startup request arrived
partner_LU_name_length	The length of *partner_LU_name*. If a blank *sym_dest_name* was specified, *partner_LU_name_length* will be 1.	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	Not applicable
send_type	CM_BUFFER_DATA	CM_BUFFER_DATA
sync_level	CM_NONE	The value received on the conversation startup request
TP_name	The program name from side information referenced by *sym_dest_name*. If a blank *sym_dest_name* was specified, *TP_name* will be a single blank.	Not applicable
TP_name_length	The length of *TP_name*. If a blank *sym_dest_name* was specified, *TP_name_length* will be 1.	Not applicable

Note: When the local program issues an Initialize_Conversation call, 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 startup 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, "Call 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 affect the initial system values (in the case of values derived from the side information).

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 call.

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 on this type of checking are provided in the individual call descriptions in Chapter 4, "Call Reference Section."

Automatic Conversion of Characteristics

Some conversation characteristics affect only the function of the local transaction program; the remote program is not aware of their settings. An example of this kind of conversation characteristic is *receive_type*. Other conversation characteristics, however, are transmitted to the remote program or LU and, thus, affect both ends of the conversation. For example, the local LU transmits the *TP_name* characteristic to the remote LU as part of the conversation startup process.

LU 6.2 protocols require that these transmitted characteristics be encoded as EBCDIC characters. For this reason, CPI Communications automatically converts these characteristics to EBCDIC when they are used as parameters on CPI Communications calls on non-EBCDIC systems. This means programmers can use the native encoding of the local system when specifying these characteristics on Set calls. Likewise, when these characteristics are returned by Extract calls, they are represented in the local system's native encoding.

The following conversation characteristics may be automatically converted by CPI Communications:

log_data	Specified on the Set_Log_Data call.
mode_name	Specified on the Extract_Mode_Name and Set_Mode_Name calls.
partner_LU_name	Specified on the Extract_Partner_LU_Name and Set_Partner_LU_Name calls.
TP_name	Specified on the Set_TP_Name call.

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 (or its equivalent) 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 control on the same Receive call. See "Example 7: Error Direction and Send-Pending State" on page 48 for a discussion of the **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's end of the conversation enters **Receive** state.
Confirm-Send	A confirmation request and send control 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's end of the conversation 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 the local program responds with Confirmed, the conversation is deallocated.

A conversation starts out in **Reset** state and moves into other states, 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.

Note: CPI Communications keeps track of a conversation's current state, as should the program. If a program issues a CPI Communications call for a conversation that is not in a valid state for the call, CPI Communications will detect this error and return a *return_code* value of CM_PROGRAM_STATE_CHECK.

The following additional states are required for programs using *sync_level*=CM_SYNC_POINT:

- **Defer-Receive**
- **Defer-Deallocate**
- **Sync-Point**
- **Sync-Point-Send**
- **Sync-Point-Deallocate**.

"Support for the SAA Resource Recovery Interface" on page 25 discusses synchronization point processing and describes these additional states.

For a complete listing of program calls, 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 enhance 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: To aid in recognizing actual call names, all CPI Communications calls begin with the prefix **CM**.

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

- **Calls** begin with capital letters, and so do the underscore-separated portions of a call's pseudonym. For example, Accept_Conversation is the pseudonym for the actual call name 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.

- **Values** used for characteristics and variables appear in all 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 5 on page 148 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, "Call 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 and 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, which goes into the Set_Return_Control call as a parameter, is a 32-bit integer. This information is provided in Table 7 on page 153.
- Table 5 on page 148 in Appendix A, "Variables and Characteristics" shows that CM_IMMEDIATE is defined as having an integer value of 1. CM_IMMEDIATE is placed into the *return_control* parameter on the call to CMSRC.
- Finally, the *return_code* value CM_OK, which is returned to the program on the CMSRC call, is defined in Appendix B, "Return Codes." CM_OK 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 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 59 in Chapter 4, "Call Reference Section" provides information on system files that can be used to establish pseudonyms for a program. See Appendix L, "Pseudonym Files" on page 335 for sample pseudonym files for the SAA programming languages that support CPI Communications.

Multiple Conversations

CPI Communications allows a program to communicate with multiple partners using multiple, concurrent conversations. For example, a program may initiate multiple conversations (using multiple Initialize_Conversation and Allocate calls), or a program may accept a conversation (using the Accept_Conversation call) and then initiate one or more additional conversations. During communications with multiple partners, the program uses the *conversation_ID* parameter, present on all CPI Communications calls, to specify the conversation for which a particular call is issued.

Figure 3 shows a program, Program A, that has established conversations with two partners. The conversation with Program B is initialized with an Initialize_Conversation (CMINIT) call that returns a *conversation_ID* of X, and the conversation with Program C is initialized with an Initialize_Conversation call that returns a *conversation_ID* of Y. When Program A issues subsequent calls with a *conversation_ID* of X, CPI Communications will know these calls apply to the conversation with Program B. Similarly, when Program A issues subsequent calls with a *conversation_ID* of Y, CPI Communications will know these calls apply to the conversation with Program C.

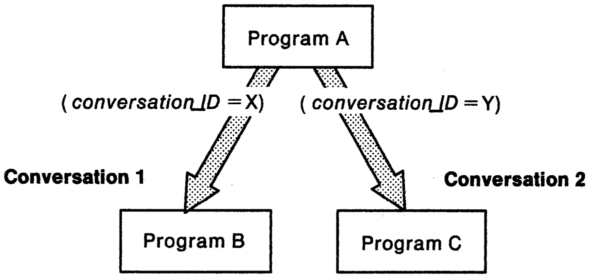


Figure 3. A Program Using Multiple CPI Communications Conversations

The default *receive_type* characteristic of CM_RECEIVE_AND_WAIT may not be desirable for a program with multiple, concurrent conversations. When the program issues a Receive call on a conversation with the *receive_type* set to CM_RECEIVE_AND_WAIT, the program goes into a wait state until data is received from its conversation partner. While in this wait state, the program is unable to perform other processing.

A program that manages multiple, concurrent conversations may not want to be placed in a wait state, from which it is unable to communicate with any of its other partners. In that case, the program can issue a Set_Receive_Type call for each concurrent conversation to change the *receive_type* to CM_RECEIVE_IMMEDIATE. Then conversations can be periodically checked for data without placing the program in a wait state.

When a program uses multiple conversations with *sync_level* set to CM_SYNC_POINT, all of these conversations will be affected if the program is placed in the **Backout-Required** condition. See "The Backout-Required Condition" on page 27 for more information.

For more information about establishing and managing multiple conversations, see the Usage Notes under "Initialize_Conversation (CMINIT)" on page 90.

Support for the SAA Resource Recovery Interface

This section describes how application programs can use CPI Communications in conjunction with the resource recovery element of the SAA common programming interface. The resource recovery interface is the SAA common programming interface to services and facilities that use two-phase commit protocols to coordinate changes to distributed resources. For information about how to use the SAA resource recovery interface in a particular operating environment, see *SAA Common Programming Interface: Resource Recovery Reference* and read the documentation for that operating environment.

Notes:

1. The following discussion is intended for programmers using CPI Communications advanced functions. Readers not interested in a high degree of synchronization need not read this section.

2. The information in this section does not apply for the CICS, IMS, MVS, OS/2, and OS/400 environments, since these products do not support use of the SAA resource recovery interface with CPI Communications.

A CPI Communications conversation can be used with the resource recovery interface only if its *sync_level* characteristic is set to CM_SYNC_POINT. This kind of conversation is called a **protected conversation.**

Coordination with the SAA Resource Recovery Interface

A program communicates with the resource recovery interface by establishing **synchronization points**, or **sync points**, in the program logic. A sync point is a reference point during transaction processing to which resources can be restored if a failure occurs. The program uses an SAA resource recovery interface Commit call to establish a new sync point or an SAA resource recovery interface Backout call to return to a previous sync point.

In turn, the resource recovery interface invokes a component of the operating environment called a **sync point manager (SPM).** The SPM coordinates the commit or backout processing among all the protected resources involved in the sync point transaction.

For more information about synchronization point processing, see *SAA Common Programming Interface: Resource Recovery Reference*.

Take-Commit and Take-Backout Notifications

When a program issues a Commit or Backout call, CPI Communications cooperates with the SAA resource recovery interface by passing synchronization information to its conversation partner. This sync point information consists of take-commit and take-backout notifications.

When the program issues a Commit call, CPI Communications returns a **take-commit notification** to the partner program in the *status_received* parameter for a Receive call issued by the partner. The sequence of CPI Communications calls issued before the resource recovery interface Commit call determines the value of the take-commit notification returned to the partner program. In addition to requesting that the partner program establish a sync point, the take-commit notification also contains conversation state transition information.

The following list shows the *status_received* values that CPI Communications uses as take-commit notifications, the conditions under which each of the values may be received, and the state changes resulting from their receipt.

status_received Value	**Conditions for Receipt**
CM_TAKE_COMMIT	The partner program issued the resource recovery interface Commit call while its end of the conversation was in **Send** state. The local program's end of the conversation is in **Sync-Point** state and will be placed back in **Receive** state once the local program issues a successful Commit call.
CM_TAKE_COMMIT_SEND	The partner program issued the Commit call while its end of the conversation was in **Defer-Receive** state. The local program's end of the conversation is in **Sync-Point-Send** state and will be placed in **Send** state once the local program issues a successful Commit call.
CM_TAKE_COMMIT_DEALLOCATE	The partner program issued the Commit call while its end of the conversation was in **Defer-Deallocate** state. The local program's end of the conversation is in **Sync-Point-Deallocate** state and will be placed in **Reset** state once the local program issues a successful Commit call.

When the program issues a Backout call, or when a system failure or a problem with a protected resource causes the SPM to initiate a backout operation, CPI Communications returns a **take-backout notification** to the partner program. CPI Communications returns this notification as one of the following values in the *return_code* parameter:

- CM_TAKE_BACKOUT
- CM_DEALLOCATED_ABEND_BO
- CM_DEALLOCATED_ABEND_SVC_BO
- CM_DEALLOCATED_ABEND_TIMER_BO
- CM_RESOURCE_FAILURE_NO_RETRY_BO
- CM_RESOURCE_FAILURE_RETRY_BO
- CM_DEALLOCATED_NORMAL_BO.

CPI Communications can return a take-backout notification on any of the following calls issued by the partner program:

- Confirm
- Extract_Conversation_State
- Prepare_To_Receive
- Receive
- Send_Data
- Send_Error.

The Backout-Required Condition

Upon receipt of a take-backout notification on one of its protected conversations, a program is in the **Backout-Required** condition. This condition is not a *conversation state*, because it applies to all of the program's protected resources, possibly including multiple conversations. When a program is in the **Backout-Required** condition, there are restrictions on the CPI Communications calls that may be used on protected conversations.

A program may be placed in the **Backout-Required** condition in one of the following ways:

- When CPI Communications returns a take-backout notification
- When the program issues a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND, or when it issues a Send_Data call with *send_type* set to CM_SEND_AND_DEALLOCATE and *deallocate_type* set to CM_DEALLOCATE_ABEND. When one of these calls is successfully issued, CPI Communications places the program in the **Backout-Required** condition.

When placed in the **Backout-Required** condition, the program should issue the SAA resource recovery interface Backout call. Until it issues a Backout call, the program will be unable to successfully issue any of the following CPI Communications calls for any of its conversations with *sync_level* set to CM_SYNC_POINT. If the program issues any of these calls, the CM_PROGRAM_STATE_CHECK return code will be returned:

- Allocate
- Confirm
- Confirmed
- Flush
- Prepare_To_Receive
- Receive
- Request_To_Send
- Send_Data
- Send_Error
- Test_Request_To_Send_Received.

Responses to Take-Commit and Take-Backout Notifications

A program usually issues a Commit or Backout call in response to a take-commit notification, and a Backout call in response to a take-backout notification. In some cases, however, the program may respond to one of these notifications with a CPI Communications call instead of a Commit or Backout call. Table 4 on page 28 shows the calls a program can use to respond to take-commit and take-backout notifications, the result of issuing each call, and any further action required by the program.

Table 4. Responses to Take-Commit and Take-Backout Notifications

Notification Received	**Possible Response**	**Reason for Response**	**Result of Response**	**Further Action Required**
Take-Commit	Commit	The program agrees that it can commit (or has committed) all protected resources.	The commit request is spread to other programs in the transaction.	None
	Backout	The program disagrees with the commit request.	A backout request is spread to other programs in the transaction, including the program that issued the original Commit call.	None
	Deallocate (Abend)[1]	The program has detected an error condition that prevents it from continuing normal processing.	The program is placed in the **Backout-Required** condition.	The program should issue the resource recovery interface Backout call.
	Send_Error	The program has detected an error in received data or some other error that may be correctable.	The SPM backs out the transaction, and both programs are informed of the backout.	Depends on the response from the partner program.
Take-Backout	Commit	This is an error in program logic.	The Commit call is treated as though it were a Backout call, and the backout request is spread to other programs in the transaction.	None
	Backout	The program agrees to the backout request.	The backout request is spread to other programs in the transaction.	None
	Deallocate (Abend)[1]	The program has detected an error condition that prevents it from continuing normal processing.	The program is placed in the **Backout-Required** condition.	The program should issue the resource recovery interface Backout call.

[1] "Deallocate (Abend)" refers to the CPI Communications Deallocate call with a *deallocate_type* of CM_DEALLOCATE_ABEND.

Additional CPI Communications States

In addition to the conversation states described in "Program Flow — States and Transitions" on page 21, the following states are required when a program uses a protected CPI Communications conversation (that is, with the *sync_level* characteristic set to CM_SYNC_POINT):

State	**Description**
Defer-Receive	The local program's end of the conversation will enter **Receive** state after a synchronization call completes successfully. The synchronization call may be the SAA resource recovery interface Commit call or a CPI Communications Flush or Confirm call. A conversation enters **Defer-Receive** state when the local program issues the Prepare_To_Receive call with *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT, or it issues a Send_Data call with *send_type* set to CM_SEND_AND_PREP_TO_RECEIVE, *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, and *sync_level* set to CM_SYNC_POINT.
Defer-Deallocate	The local program has requested that the conversation be deallocated after a commit operation has completed; that is, it issued a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT, or it issued a Send_Data call with *send_type* set to CM_SEND_AND_DEALLOCATE, *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL, and *sync_level* set to CM_SYNC_POINT. The conversation will not be deallocated until a successful commit operation takes place.
Sync-Point	The local program issued a Receive call and was given a *return_code* of CM_OK and a *status_received* of CM_TAKE_COMMIT. After a successful commit operation, the conversation will return to Receive state.
Sync-Point-Send	The local program issued a Receive call and was given a *return_code* of CM_OK and a *status_received* of CM_TAKE_COMMIT_SEND. After a successful commit operation, the conversation will be placed in **Send** state.
Sync-Point-Deallocate	The local program issued a Receive call and was given a *return_code* of CM_OK and a *status_received* of CM_TAKE_COMMIT_DEALLOCATE. After a successful commit operation, the conversation will be deallocated and placed in **Reset** state.

Valid States for Commit Calls

A program must be in one of the following conversation states to issue the SAA resource recovery interface Commit call:

- **Reset**
- **Initialize**
- **Send**
- **Send-Pending**
- **Defer-Receive**
- **Defer-Deallocate**
- **Sync-Point**
- **Sync-Point-Send**
- **Sync-Point-Deallocate.**

If a Commit call is issued from any other conversation state, the program receives an SAA resource recovery interface return code of RR_PROGRAM_STATE_CHECK. The program can also receive this return code if the conversation was in **Send** state when the Commit call was issued, and the program had started but had not finished sending a basic conversation logical record.

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 32
- "Advanced-Function Flows" on page 39.

In addition to these sample flows, a simple COBOL application using CPI Communications calls is provided in Appendix K, "Sample Programs" on page 323.

A Word about the Flow Diagrams

In the flow diagrams shown in this chapter (for example, Figure 4 on page 35), 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, "Call 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, "Call Reference Section."

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 33 demonstrates a flow of data in only one direction (only the initiating program sends data).
- "Example 2: Data Flow in Both Directions" on page 36 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 4 on page 35 shows an example of a conversation where the flow of data is in only one direction.

The steps shown in Figure 4 are:

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 assigns a *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 90, have been assigned. (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 67) 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 67 for more information about the error conditions that can result.

6 and 7 Program A sends data with the Send_Data call (described in "Send_Data (CMSEND)" on page 107) and receives a *return_code* of CM_OK. Until now, the conversation may not have been established because the conversation startup 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 startup 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 startup 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 39.

Step	Description
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 65 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 18, some of Program C's defaults are based on information contained in the conversation startup request.
10 and 11	Once its end of the conversation is 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 97). Program A could continue to make Send_Data calls (and Program C could continue to make Receive calls), but, for the purposes of this example, assume that Program A only wanted to send the data contained in its initial Send_Data call.
12	Program A issues a Deallocate call (see "Deallocate (CMDEAL)" on page 75) 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.

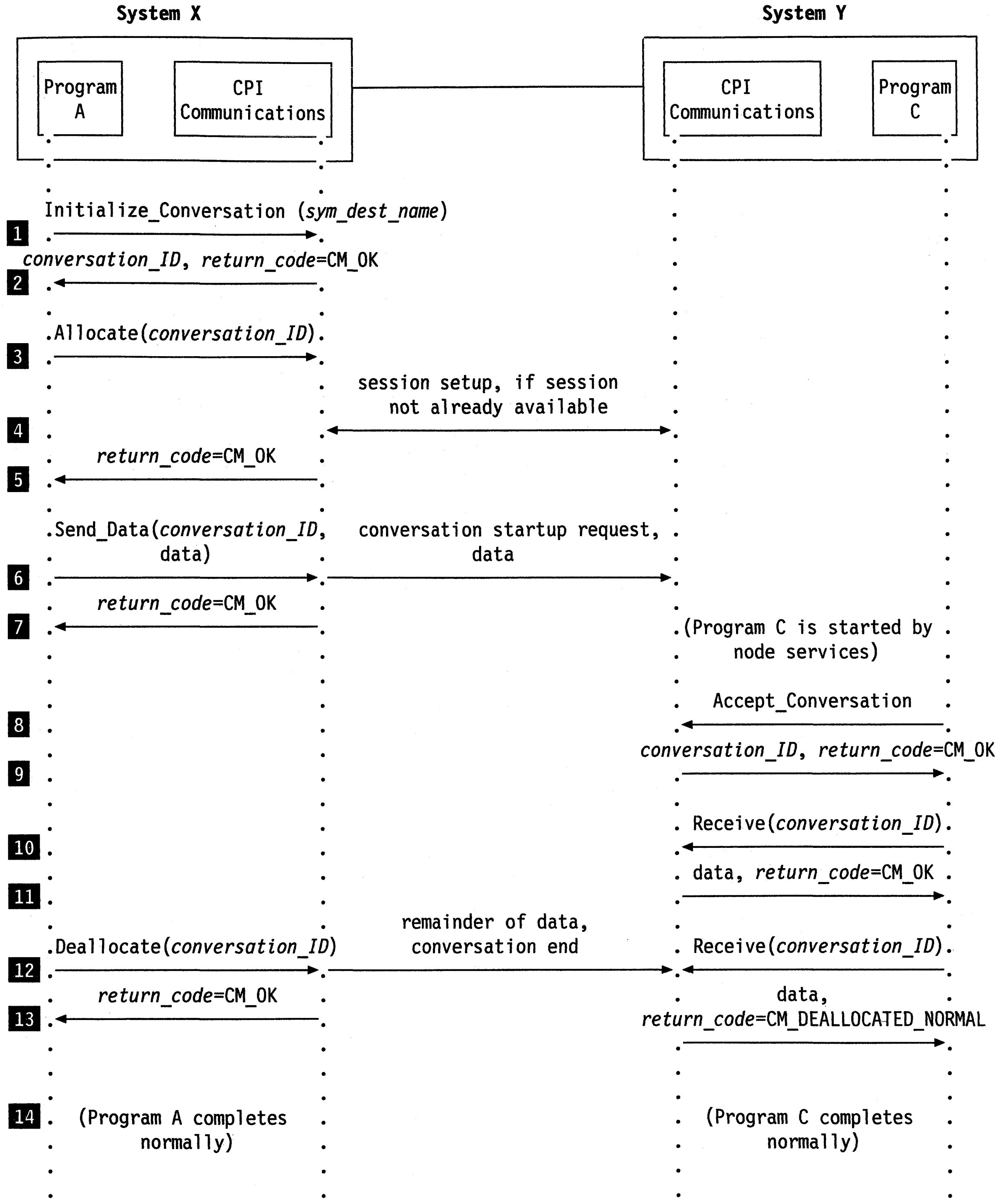

System X
System Y
Program A
CPI Communications
CPI Communications
Program C
Initialize_Conversation (*sym_dest_name*)
1
conversation_ID, *return_code*=CM_OK
2
Allocate(*conversation_ID*)
3
session setup, if session not already available
4
return_code=CM_OK
5
Send_Data(*conversation_ID*, data)
conversation startup request, data
6
return_code=CM_OK
7
(Program C is started by node services)
Accept_Conversation
8
conversation_ID, *return_code*=CM_OK
9
Receive(*conversation_ID*)
10
data, *return_code*=CM_OK
11
remainder of data, conversation end
Deallocate(*conversation_ID*)
Receive(*conversation_ID*)
12
return_code=CM_OK
data, *return_code*=CM_DEALLOCATED_NORMAL
13
14
(Program A completes normally)
(Program C completes normally)

Figure 4. Data Flow in One Direction

Example 2: Data Flow in Both Directions

Figure 5 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 5 are:

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 its end of the conversation is in **Send** state. As described in "Receive (CMRCV)" on page 97, 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's end of the conversation is placed in **Receive** state, and Program A waits for a response from Program C. **Note:** See "Example 3: The Sending Program Changes the Data Flow Direction" on page 40 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 its end of 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	Programs A and C continue processing. Program C sends data and Program A receives the data.

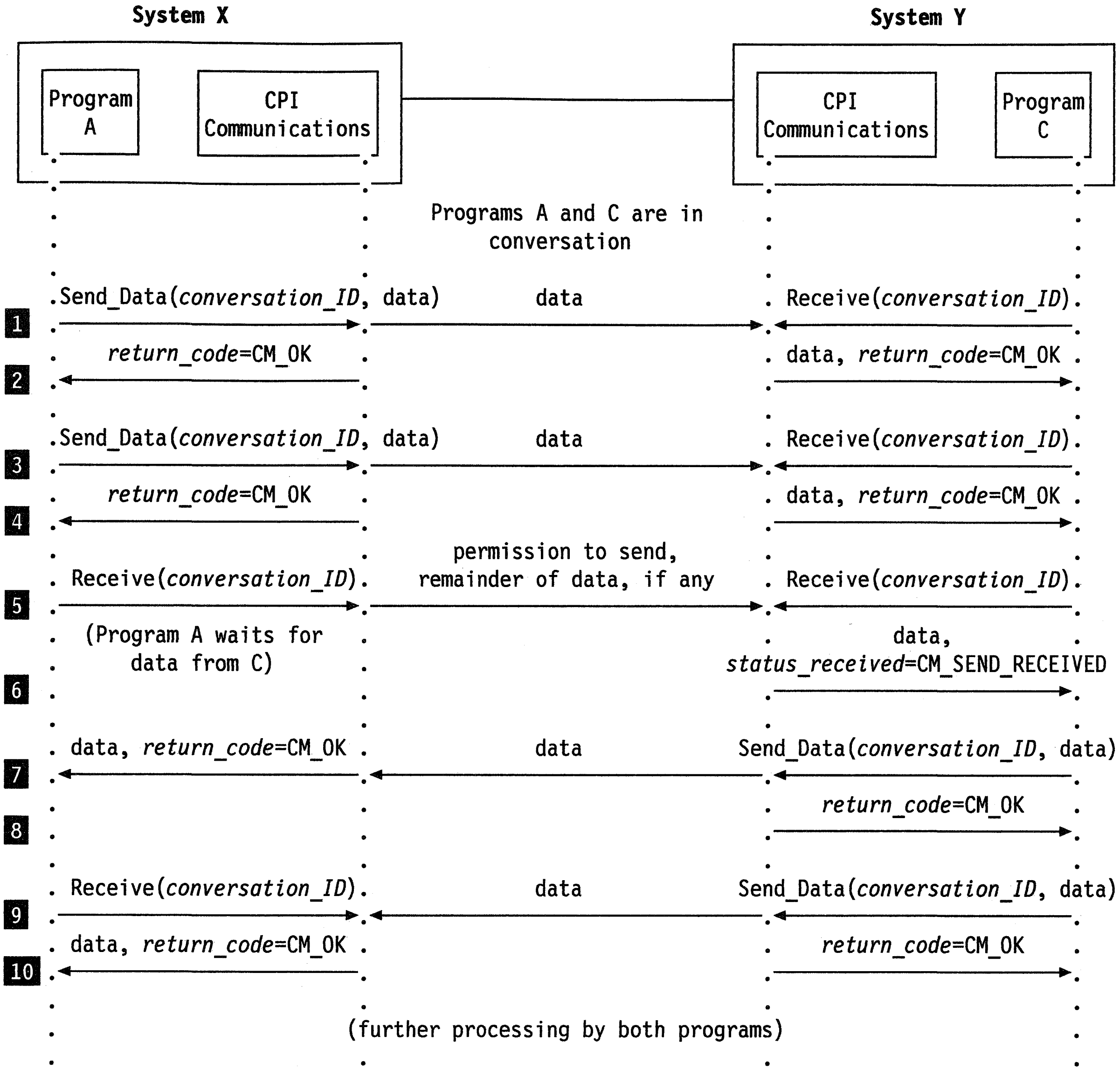

System X
System Y
Program A
CPI Communications
CPI Communications
Program C
Programs A and C are in conversation
1
Send_Data(*conversation_ID*, data)
data
Receive(*conversation_ID*)
2
return_code=CM_OK
data, *return_code*=CM_OK
3
Send_Data(*conversation_ID*, data)
data
Receive(*conversation_ID*)
4
return_code=CM_OK
data, *return_code*=CM_OK
5
Receive(*conversation_ID*)
permission to send, remainder of data, if any
Receive(*conversation_ID*)
(Program A waits for data from C)
6
data, *status_received*=CM_SEND_RECEIVED
7
data, *return_code*=CM_OK
data
Send_Data(*conversation_ID*, data)
8
return_code=CM_OK
9
Receive(*conversation_ID*)
data
Send_Data(*conversation_ID*, data)
10
data, *return_code*=CM_OK
return_code=CM_OK
(further processing by both programs)

Figure 5. 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 the direction of the 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 the 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.
- "Example 8: Sending Program Issues a Commit" shows how to use a protected conversation with an SAA resource recovery interface Commit call to establish a synchronization point for protected resources.
- "Example 9: A Successful Commit with Conversation State Change" shows a successful SAA resource recovery interface Commit operation with a conversation state change.
- "Example 10: Conversation Deallocation before the Commit Call" shows an SAA resource recovery interface Commit call issued after a CPI Communications Deallocate 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 startup of the conversation and subsequent data flow can occur anytime 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 and the use of Deallocate have already been shown in "Starter-Set Flows" on page 32. The other calls are discussed in the following examples.

Example 3: The Sending Program Changes the Data Flow Direction

Figure 6 is a variation on the function provided by the flow shown in "Example 2: Data Flow in Both Directions" on page 36. When the data flow direction changes, Program A can continue processing instead of waiting for data to arrive.

The steps shown in Figure 6 are:

Step	Description
1 through 6	The program begins the same as "Example 1: Data Flow in One Direction" on page 33. 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 93), 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's end of the conversation 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 startup 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. 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 Receive calls 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. This method of receiving data is not shown in Figure 6. For further discussion of this alternate flow, see "Set_Receive_Type (CMSRT)" on page 135 and "Receive (CMRCV)" on page 97.

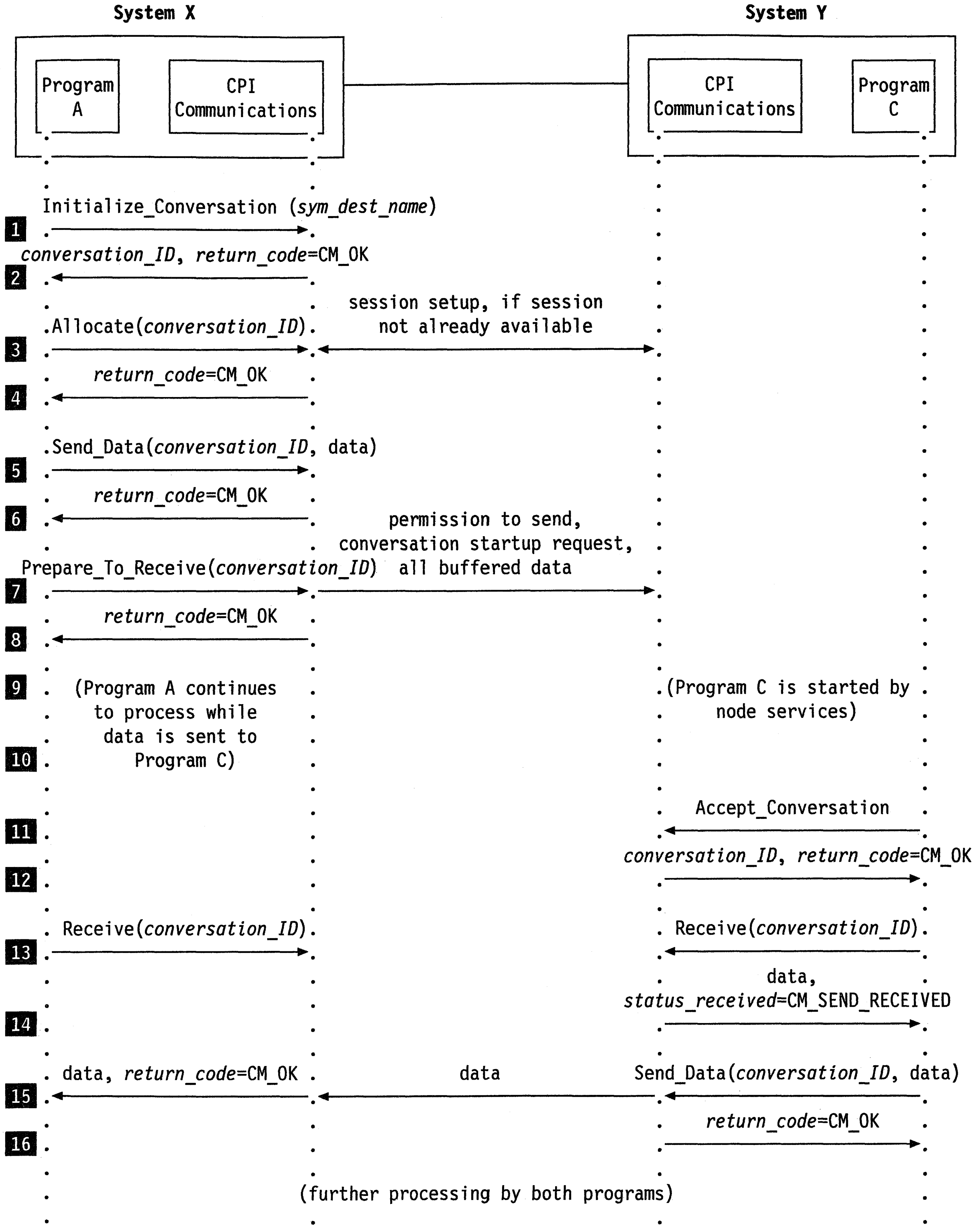

System X
System Y
Program A
CPI Communications
CPI Communications
Program C
1
Initialize_Conversation (*sym_dest_name*)
2
conversation_ID, *return_code*=CM_OK
3
Allocate(*conversation_ID*)
session setup, if session not already available
4
return_code=CM_OK
5
Send_Data(*conversation_ID*, data)
6
return_code=CM_OK
7
Prepare_To_Receive(*conversation_ID*)
permission to send, conversation startup request, all buffered data
8
return_code=CM_OK
9
(Program A continues to process while data is sent to Program C)
10
(Program C is started by node services)
11
Accept_Conversation
12
conversation_ID, *return_code*=CM_OK
13
Receive(*conversation_ID*)
Receive(*conversation_ID*)
14
data, *status_received*=CM_SEND_RECEIVED
15
data, *return_code*=CM_OK
data
Send_Data(*conversation_ID*, data)
16
return_code=CM_OK
(further processing by both programs)

Figure 6. The Sending Program Changes the Data Flow Direction

Example 4: Validation and Confirmation of Data Reception

Figure 7 on page 43 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 7 are:

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, to set the *sync_level* characteristic to CM_CONFIRM. **Note:** Program A must set the *sync_level* characteristic before issuing the Allocate call (Step 5) for the value to take effect. Attempting to change the *sync_level* after the conversation is allocated results in an error condition. See "Set_Sync_Level (CMSSL)" on page 141 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 88) to make sure that the conversation is immediately established. If data is present, the local LU buffer is emptied and the contents are sent to the remote LU. Since no data is present, only the LU 6.2 conversation startup request is sent to establish the conversation. At System Y, the conversation startup 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 73), 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.

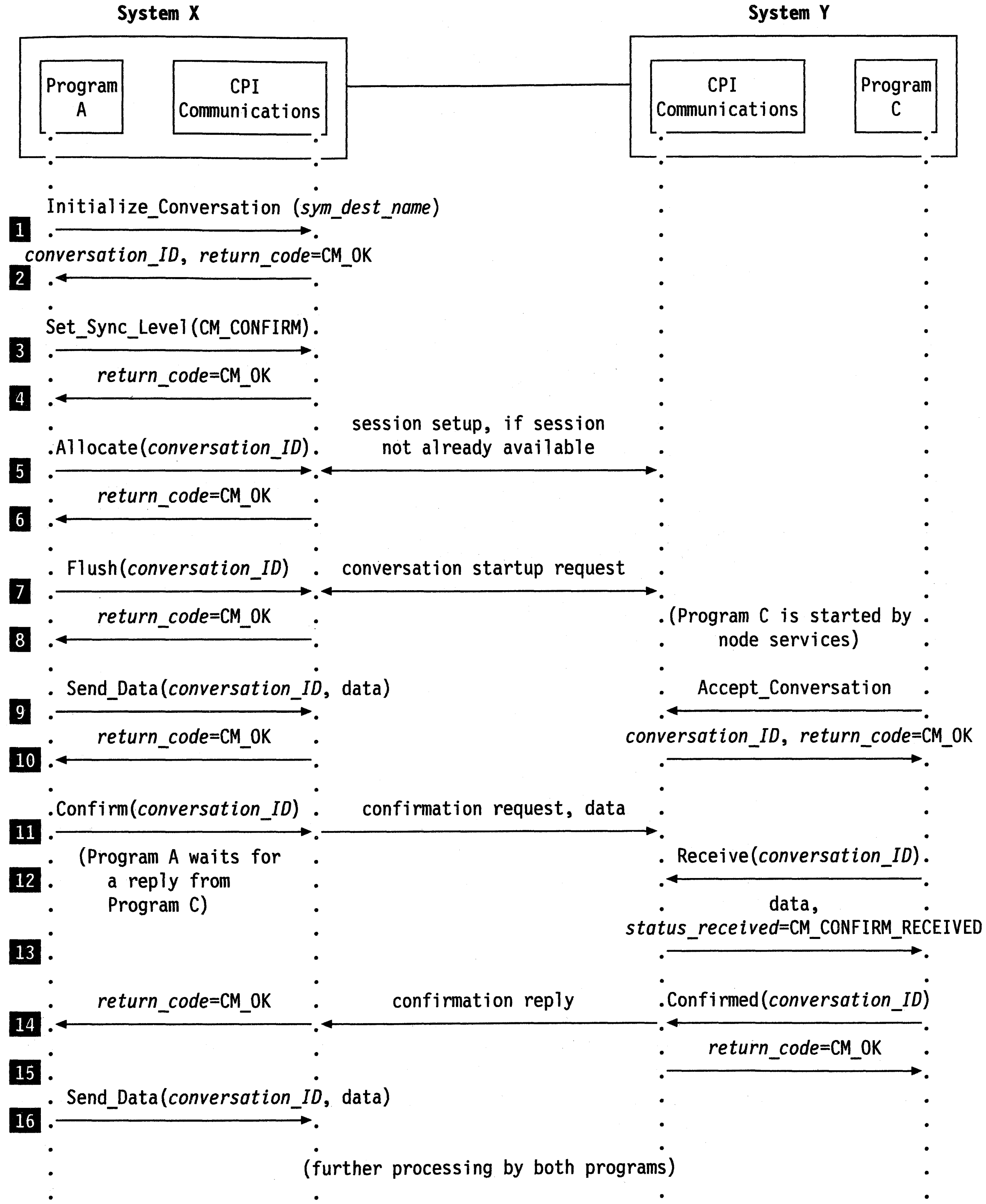

System X
System Y
Program A
CPI Communications
CPI Communications
Program C
Initialize_Conversation (sym_dest_name)
1
conversation_ID, return_code=CM_OK
2
Set_Sync_Level(CM_CONFIRM)
3
return_code=CM_OK
4
Allocate(conversation_ID)
session setup, if session not already available
5
return_code=CM_OK
6
Flush(conversation_ID)
conversation startup request
7
return_code=CM_OK
(Program C is started by node services)
8
Send_Data(conversation_ID, data)
Accept_Conversation
9
return_code=CM_OK
conversation_ID, return_code=CM_OK
10
Confirm(conversation_ID)
confirmation request, data
11
(Program A waits for a reply from Program C)
Receive(conversation_ID)
12
data, status_received=CM_CONFIRM_RECEIVED
13
return_code=CM_OK
confirmation reply
Confirmed(conversation_ID)
14
return_code=CM_OK
15
Send_Data(conversation_ID, data)
16
(further processing by both programs)

Figure 7. Validation and Confirmation of Data Reception

Example 5: The Receiving Program Changes the Data Flow Direction

Figure 8 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 105 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 8 are:

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 issued by Program A (Send_Data in this case, Step 6).
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 buffer 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 40 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's end of the conversation will return to **Receive** state after Program C issues a Send_Data call. It also forces a flushing of the LU's data buffer.
11	Program C issues the Send_Data call and its end of the conversation 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 Receive call (which is perfectly timed, in this diagram).
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.

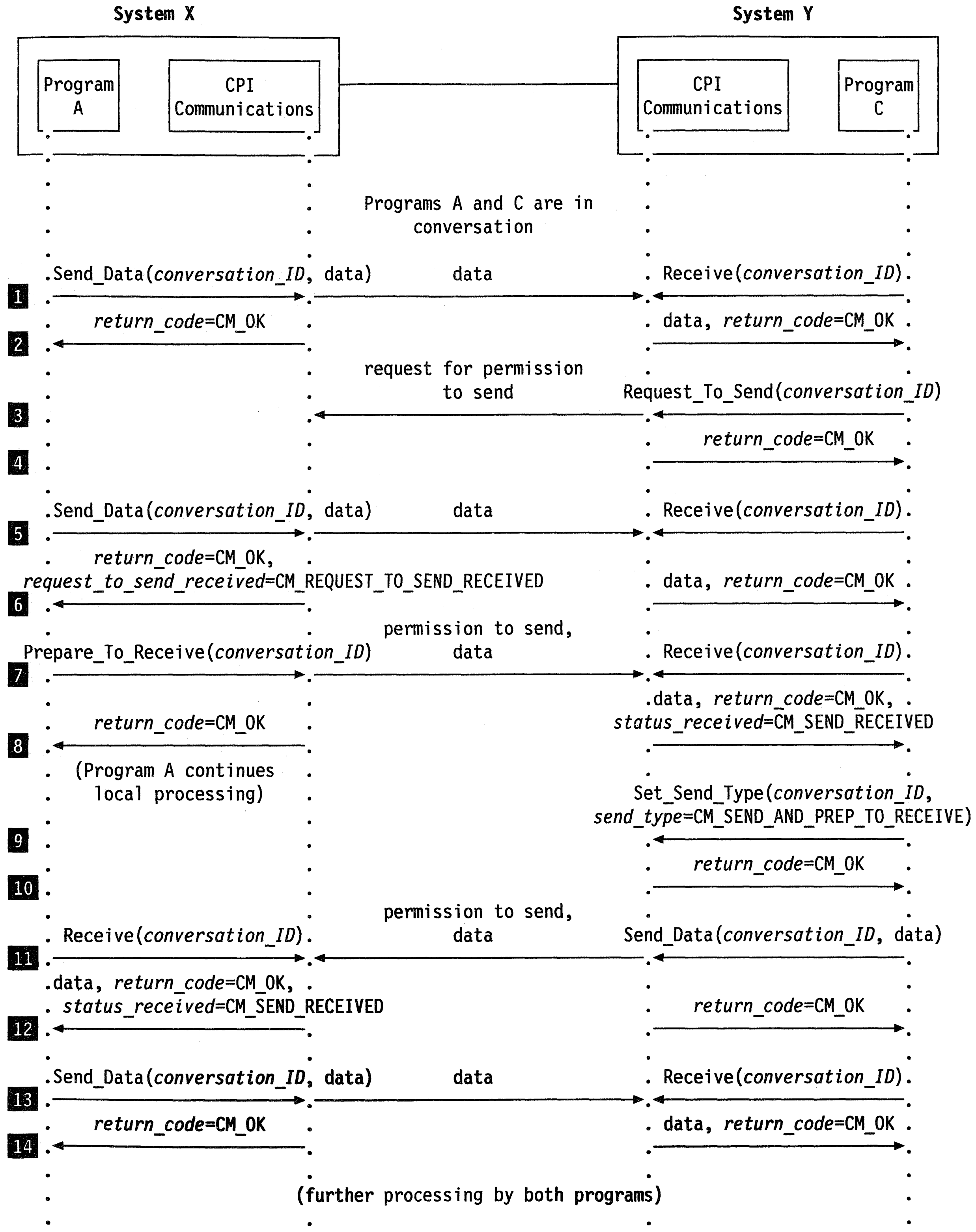

System X
System Y
Program A
CPI Communications
CPI Communications
Program C
Programs A and C are in conversation
1
Send_Data(*conversation_ID*, data)
data
Receive(*conversation_ID*)
2
return_code=CM_OK
data, *return_code*=CM_OK
3
request for permission to send
Request_To_Send(*conversation_ID*)
4
return_code=CM_OK
5
Send_Data(*conversation_ID*, data)
data
Receive(*conversation_ID*)
6
return_code=CM_OK,
request_to_send_received=CM_REQUEST_TO_SEND_RECEIVED
data, *return_code*=CM_OK
7
permission to send, data
Prepare_To_Receive(*conversation_ID*)
Receive(*conversation_ID*)
8
return_code=CM_OK
data, *return_code*=CM_OK,
status_received=CM_SEND_RECEIVED
(Program A continues local processing)
9
Set_Send_Type(*conversation_ID*,
send_type=CM_SEND_AND_PREP_TO_RECEIVE)
10
return_code=CM_OK
11
permission to send, data
Receive(*conversation_ID*)
Send_Data(*conversation_ID*, data)
12
data, *return_code*=CM_OK,
status_received=CM_SEND_RECEIVED
return_code=CM_OK
13
Send_Data(*conversation_ID*, data)
data
Receive(*conversation_ID*)
14
return_code=CM_OK
data, *return_code*=CM_OK
(further processing by both programs)

Figure 8. 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 9 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 48 describes a more complicated use of Send_Error.

The steps shown in Figure 9 are:

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 placed into **Receive** state, in a parallel action to the now-new **Send** state of the conversation for Program C.
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.

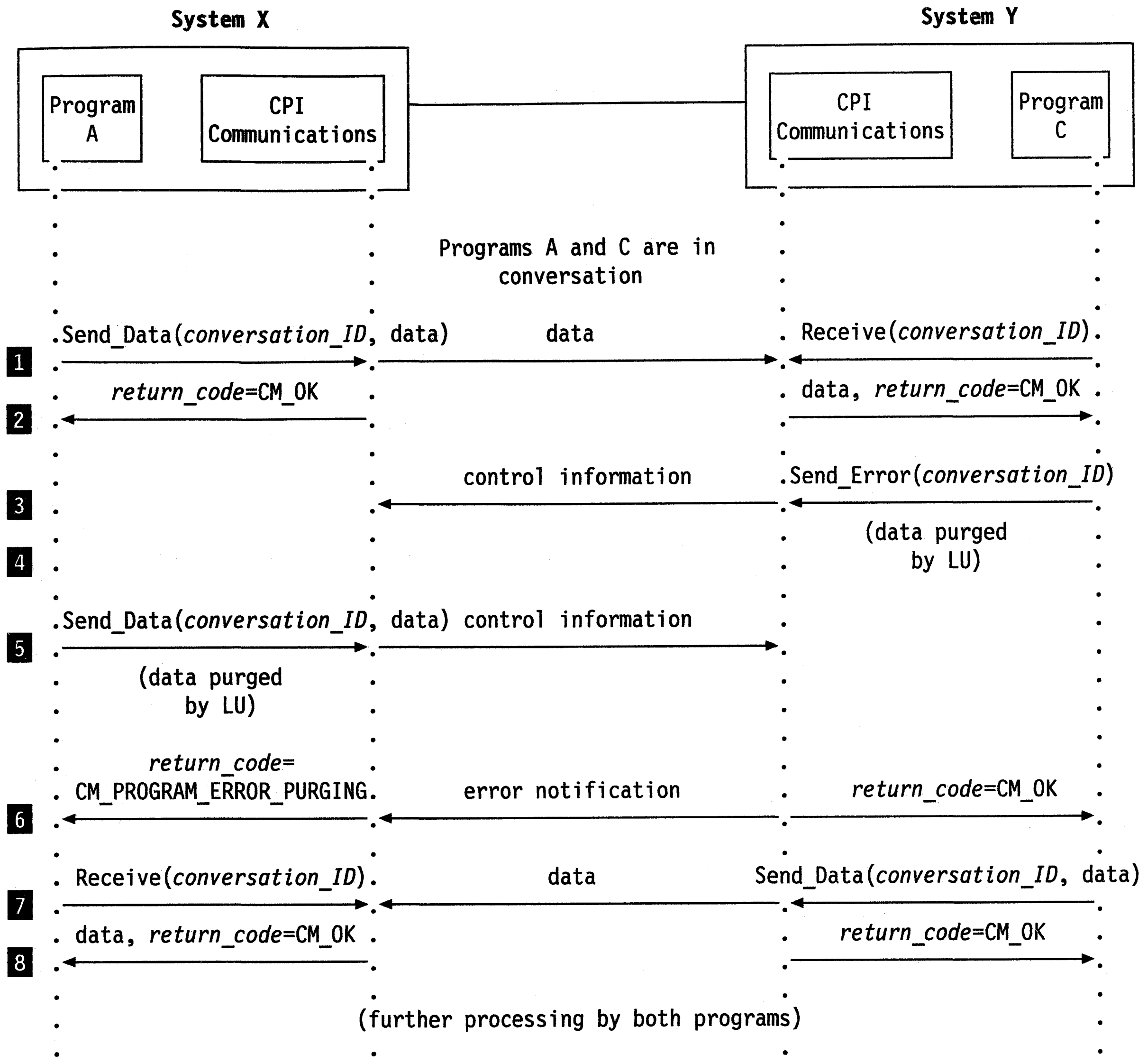

System X
System Y
Program A
CPI Communications
CPI Communications
Program C
Programs A and C are in conversation
Send_Data(*conversation_ID*, data)
data
Receive(*conversation_ID*)
1
return_code=CM_OK
data, *return_code*=CM_OK
2
control information
Send_Error(*conversation_ID*)
3
(data purged by LU)
4
Send_Data(*conversation_ID*, data)
control information
5
(data purged by LU)
return_code=
CM_PROGRAM_ERROR_PURGING
error notification
return_code=CM_OK
6
Receive(*conversation_ID*)
data
Send_Data(*conversation_ID*, data)
7
data, *return_code*=CM_OK
return_code=CM_OK
8
(further processing by both programs)

Figure 9. Reporting Errors

Example 7: Error Direction and Send-Pending State

Figure 10 on page 49 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 10 are:

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 its end of the conversation 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's end of the conversation 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 database. Program C successfully processes the query but finds that the database is not available when it attempts to access that database. 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 conversation 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 *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.

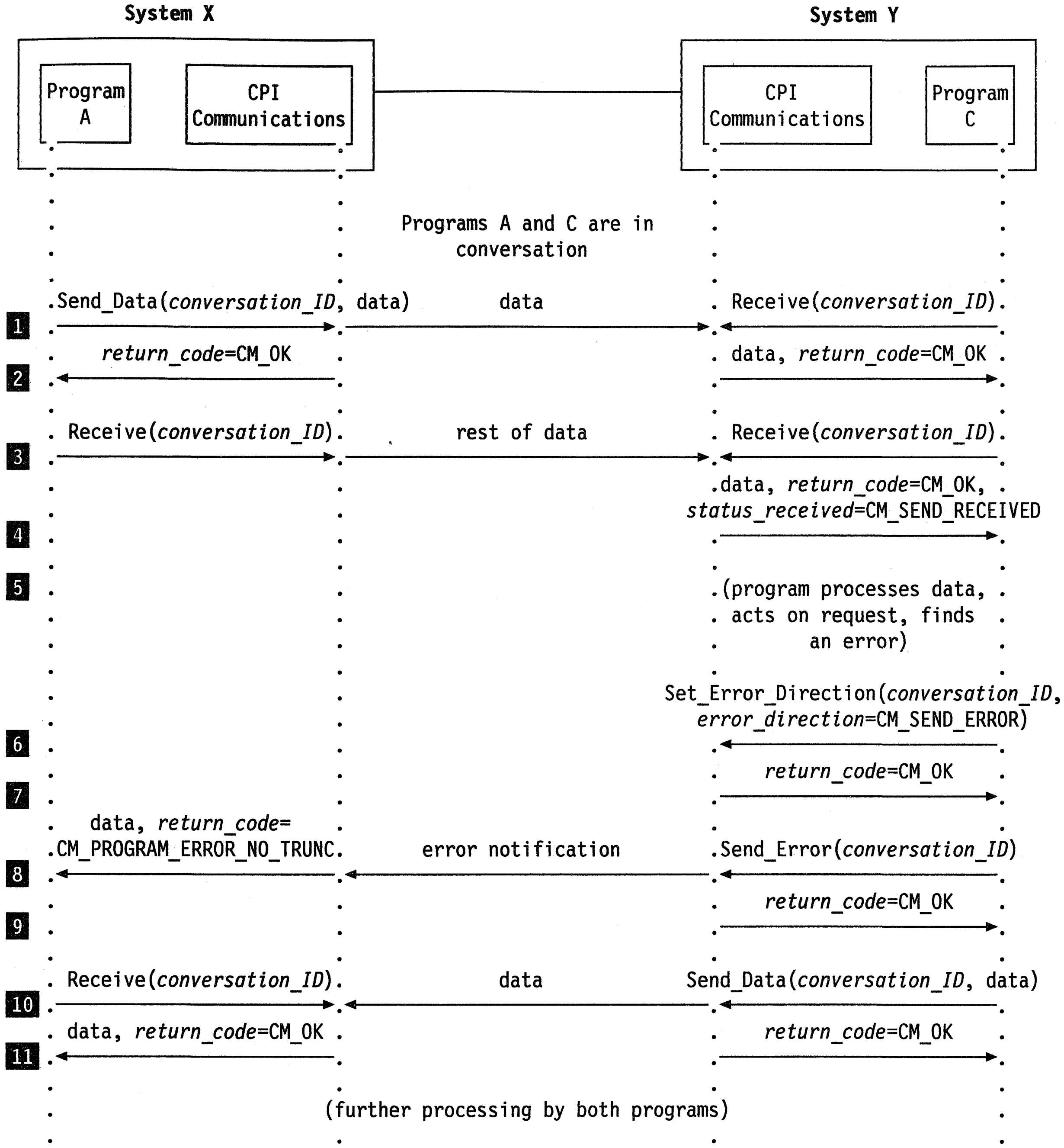

System X
System Y
Program A
CPI Communications
CPI Communications
Program C
Programs A and C are in conversation
Send_Data(*conversation_ID*, data)
data
Receive(*conversation_ID*)
1
return_code=CM_OK
data, *return_code*=CM_OK
2
Receive(*conversation_ID*)
rest of data
Receive(*conversation_ID*)
3
data, *return_code*=CM_OK, *status_received*=CM_SEND_RECEIVED
4
5
(program processes data, acts on request, finds an error)
Set_Error_Direction(*conversation_ID*, *error_direction*=CM_SEND_ERROR)
6
return_code=CM_OK
7
data, *return_code*= CM_PROGRAM_ERROR_NO_TRUNC
error notification
Send_Error(*conversation_ID*)
8
return_code=CM_OK
9
Receive(*conversation_ID*)
data
Send_Data(*conversation_ID*, data)
10
data, *return_code*=CM_OK
return_code=CM_OK
11
(further processing by both programs)

Figure 10. Error Direction and Send-Pending State

The following examples show the use of CPI Communications with the SAA resource recovery interface. Because the resource recovery interface is used to synchronize changes to all protected resources in a transaction, these examples show databases as participants in the work accomplished over CPI Communications protected conversations.

The acronym **RR/SPM** is used in the following examples to represent the resource recovery interface functioning in conjunction with a sync point manager.

Example 8: Sending Program Issues a Commit

This example shows a program sending data on a protected conversation and issuing an SAA resource recovery interface Commit call. A protected conversation is one in which the *sync_level* has been set to CM_SYNC_POINT. This synchronization level tells CPI Communications that the program will use SAA resource recovery interface calls to manage the changes made to protected resources.

The steps in Figure 11 are described below:

Step	**Description**
1 through 6	To communicate with its partner program, Program A must first establish a conversation. It uses the Set_Sync_Level call in step 3 to request that the conversation be protected.
7	Program A sends data, and Program C issues a Receive call that allows it to receive the data.
8	Both Program A and Program C get return codes indicating that their respective calls have completed successfully.
9	Program A issues a Commit call to make all of the updates permanent and to advance all of the protected resources to a new synchronization point. Program C's end of the conversation is still in **Receive** state and it issues a second Receive call.
10	On System X, a request to commit is sent from the sync point manager to the CPI Communications component. The CPI Communications component of System X propagates the request to its counterpart, the CPI Communications component of System Y, using the CPI Communications conversation. Any data remaining in Program A's send buffer is flushed at this point.
11	Program C's Receive call executes successfully and it receives the take-commit notification from the CPI Communications component of System Y.
12	Program C responds to the take-commit notification by issuing a Commit call.
13	Commit-processing protocols are exchanged between the two sync point managers.
14	Both Program A and Program C receive return codes that indicate successful completion of the Commit operation. Program A can now send more data to Program C.

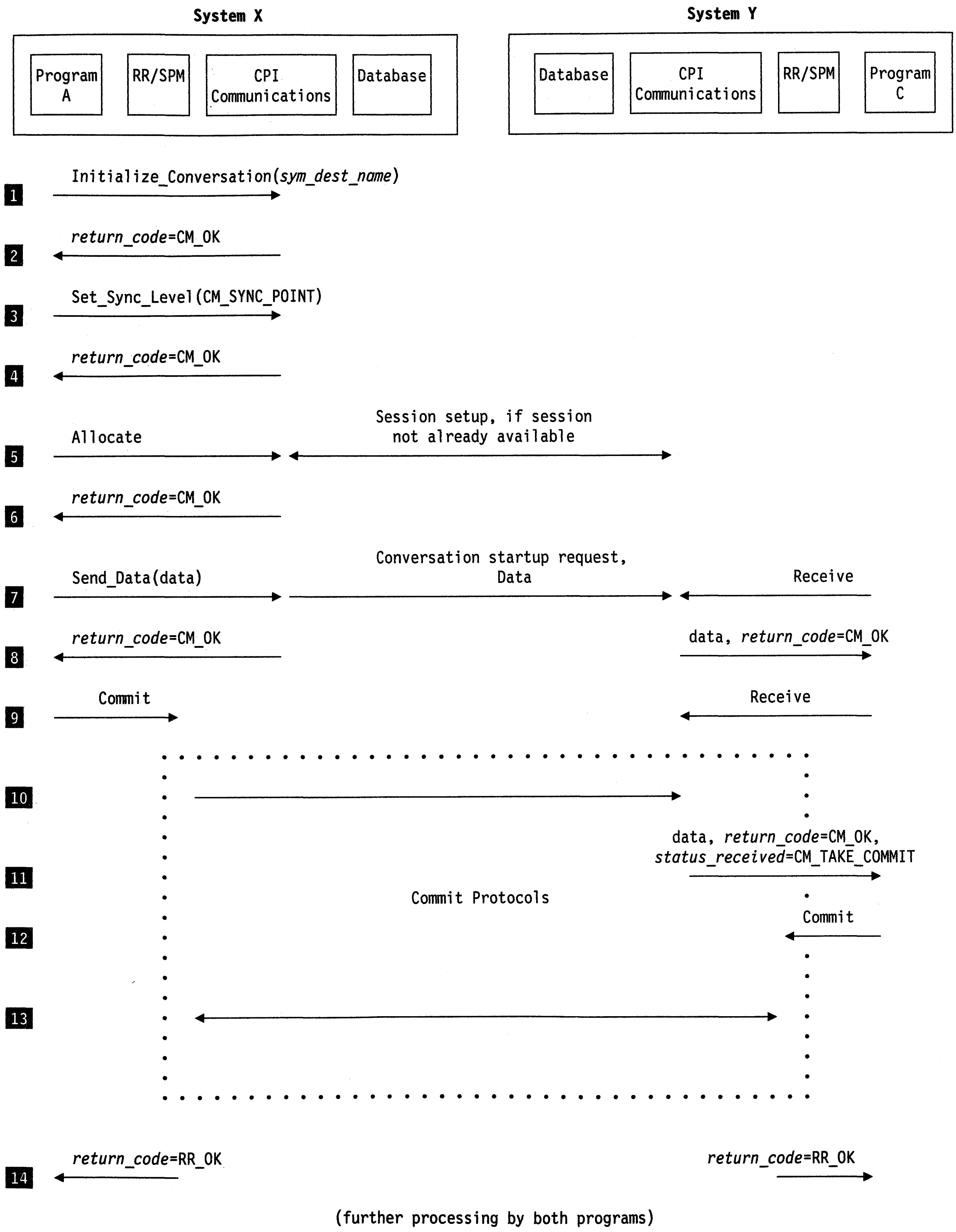


Figure 11. Establishing a Protected Conversation and Issuing a Successful Commit

Example 9: A Successful Commit with Conversation State Change

Figure 12 on page 53 shows a successful Commit with a conversation state change.

The steps in Figure 12 are described below:

Step	Description
1	Program C's end of a protected conversation is in **Receive** state. It issues a Receive call.
2 and 3	Program A wants its side of the CPI Communications conversation to be changed from **Send** to **Receive** state after it issues its next Commit call. To do this, Program A uses the Set_Prepare_To_Receive_Type call to set the *prepare_to_receive_type* conversation characteristic to CM_PREP_TO_RECEIVE_SYNC_LEVEL.
4 and 5	Program A issues a Prepare_To_Receive call. Because the *prepare_to_receive_type* conversation characteristic is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, Program A's side of the conversation is now in **Defer-Receive** state until Program A issues a Commit call.
6	Program A issues the Commit call in **Defer-Receive** state. If the call completes successfully, Program A's end of the conversation will be placed in **Receive** state.
7	The sync point manager of System X sends Program A's request to commit to the CPI Communications component of System X, which passes it to the CPI Communications component of System Y. Any data remaining in Program A's send buffer is flushed at this point.
8	Because Program A was in **Defer-Receive** state when it issued the Commit call, the CPI Communications component of System Y returns the take-commit notification to Program C as a CM_TAKE_COMMIT_SEND value in the *status_received* parameter. This value means that if Program C completes a Commit call successfully, its end of the conversation will be placed in **Send** state.
9	Program C responds to the take-commit notification with a Commit call.
10 through 12	The database managers and the sync point managers on the two systems participate in the protocol flows necessary to accomplish the commit.
13	Both Commit calls end successfully.
14	Program A's end of the conversation is now in **Receive** state and it issues a Receive call. Program C's end of the conversation is in **Send** state and it issues a Send_Data call. **Note:** If one of the following return codes were received on the Commit call by Program C, the conversation states for Programs A and C would be reset to their values at the time of the last sync point. • RR_BACKED_OUT • RR_BACKED_OUT_OUTCOME_PENDING • RR_BACKED_OUT_OUTCOME_MIXED. The program can retrieve the current conversation state by using the CPI Communications Extract_Conversation_State call.

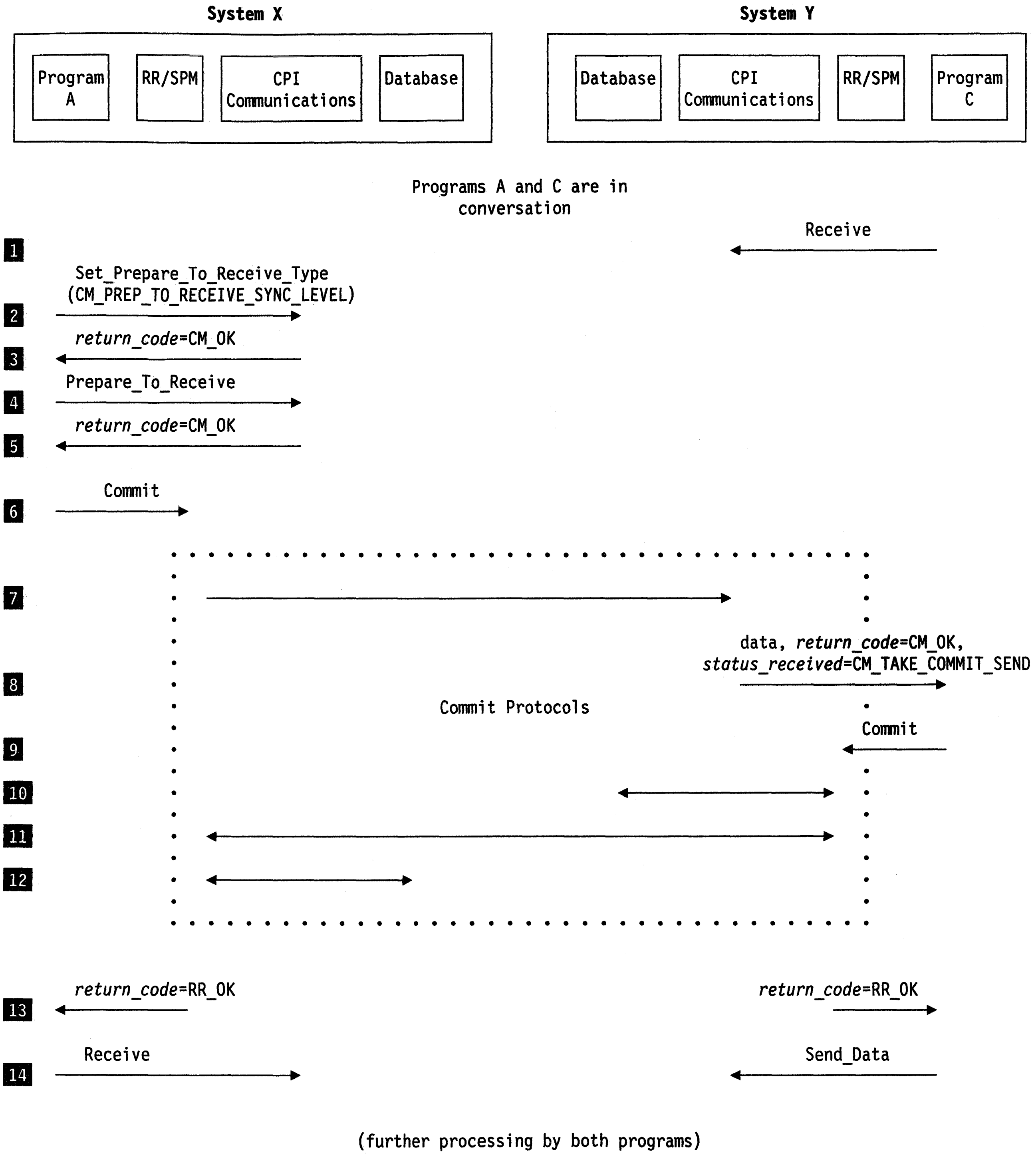


Figure 12. A Successful Commit with Conversation State Change

Example 10: Conversation Deallocation before the Commit Call

Figure 13 on page 55 shows a Commit call issued after a CPI Communications Deallocate call.

The steps in Figure 13 are described below:

Step	Description
1	Program C's end of a protected conversation is in **Receive** state. It issues a Receive call.
2 and 3	Program A wants to deallocate the conversation after issuing a Commit call. To accomplish this, Program A chooses to issue a Set_Deallocate_Type call with the *deallocate_type* parameter set to CM_DEALLOCATE_SYNC_LEVEL.
4 and 5	Program A next issues a Deallocate call. CPI Communications completes the call, and places Program A's side of the conversation in **Defer-Deallocate** state.
6	Program A issues a Commit call. If the call completes successfully, Program A's end of the conversation will be deallocated (put in **Reset** state).
7	The sync point manager of System X sends Program A's request to commit to the CPI Communications component of System X, which passes it to the CPI Communications component of System Y. Any data remaining in Program A's send buffer is flushed at this point.
8	Because Program A's end of the conversation was in **Defer-Deallocate** state when Program A issued the Commit call, Program C receives the take-commit notification as a CM_TAKE_COMMIT_DEALLOCATE value in the *status_received* parameter of its Receive call.
9	Program C responds to the take-commit notification by issuing a Commit call. The CM_TAKE_COMMIT_DEALLOCATE value means that if this Commit is successful, Program C's end of the conversation will be deallocated (put in **Reset** state).
10 through 12	The database managers and the sync point managers on the two systems participate in the protocol flows necessary to accomplish the commit.
13	Both Commit calls end successfully. **Note:** If one of the following return codes were received on the Commit call by Program C, the conversation would not be deallocated. The conversation states for Programs A and C would be reset to their values at the time of the last sync point. • RR_BACKED_OUT • RR_BACKED_OUT_OUTCOME_PENDING • RR_BACKED_OUT_OUTCOME_MIXED. The program can retrieve the current conversation state by using the CPI Communications Extract_Conversation_State call.

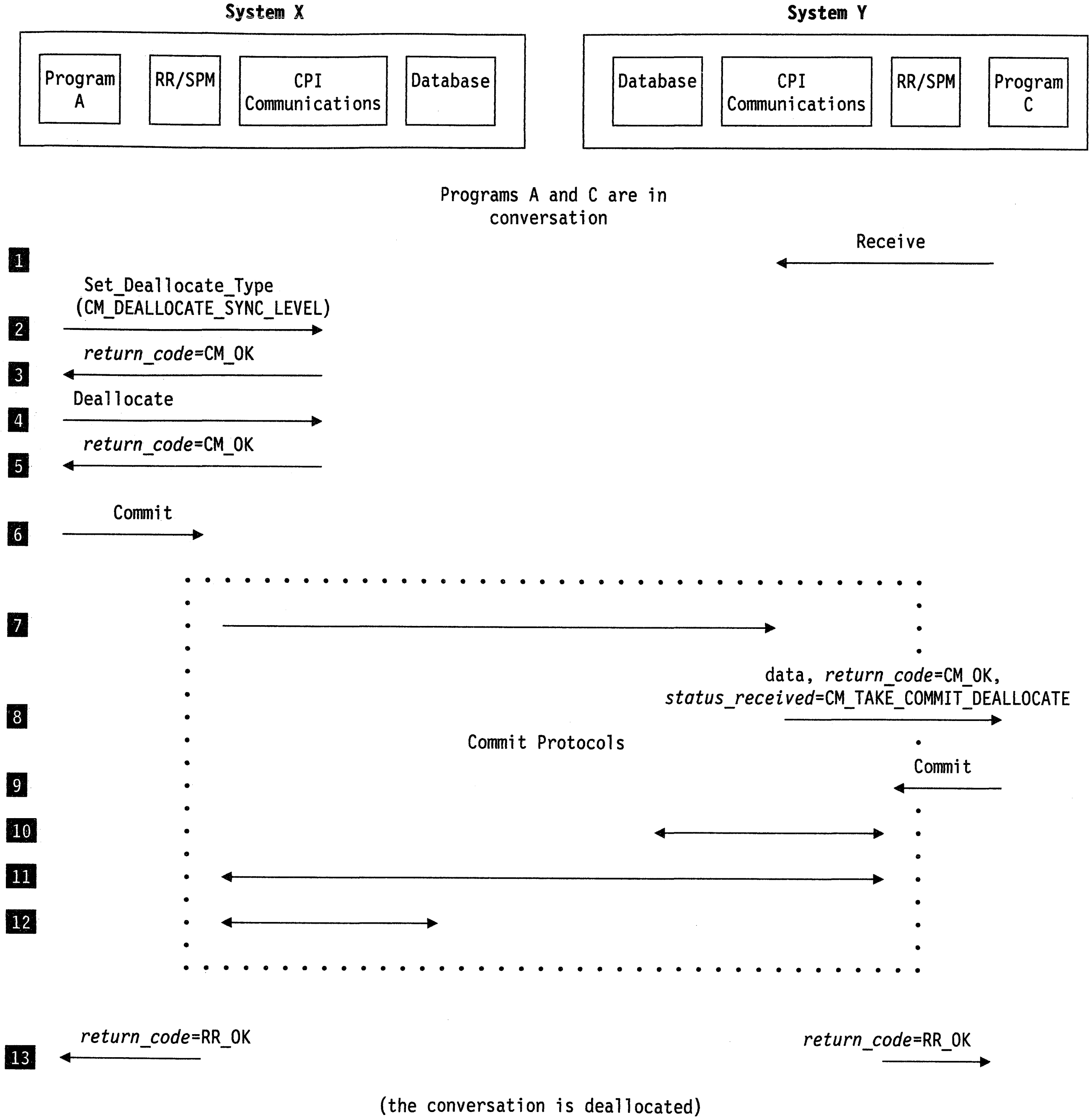


Figure 13. Conversation Deallocation Precedes the Commit Call

Chapter 4. Call 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 call.

  Note: The actual syntax used to program the calls in this chapter depends on the programming language used. See "Call Syntax" on page 58 for specifics.

- **Parameters**

  The parameters that are required for the call. Parameters are identified as *input* parameters (that is, set by the calling program and used as input to CPI Communications) or *output* parameters (that is, set by CPI Communications before returning control to the calling program).

- **State Changes**

  The changes in the conversation state that can result from this call. See "Program Flow — States and Transitions" on page 21 for more information on conversation states.

- **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:

- Application Generator
- C
- COBOL
- FORTRAN
- PL/I
- Procedures Language
- RPG.

CPI Communications calls from Application Generator programs are not currently supported on CICS or OS/2 systems.

CPI Communications calls from FORTRAN programs are not currently supported on CICS systems.

CPI Communications calls from PL/I programs are not currently supported on OS/2 or OS/400 systems.

CPI Communications calls from Procedures Language (REXX) programs are not currently supported on CICS systems.

CPI Communications calls from RPG programs are not currently supported on CICS, IMS, MVS, OS/2, or VM systems.

In addition to the above SAA languages, other languages may support CPI Communications calls in certain environments. For more information about supported languages as well as specific syntax and library information for each operating environment that implements CPI Communications, refer to the following appendixes:

CICS	Appendix E, "CPI Communications on CICS/ESA" on page 187
IMS	Appendix F, "CPI Communications on IMS/ESA" on page 193
MVS	Appendix G, "CPI Communications on MVS/ESA" on page 195
OS/2	Appendix H, "CPI Communications on OS/2" on page 201
OS/400	Appendix I, "CPI Communications on Operating System/400" on page 261
VM	Appendix J, "CPI Communications on VM/ESA CMS" on page 279.

This book uses a general call format to show the name of the CPI Communications call and the 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 description.

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

Application Generator

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

C

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

COBOL

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

FORTRAN

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

PL/I

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

Procedures Language

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

RPG

```
CALL 'CMPROG'
PARM         parm1
PARM         parm2
 .             .
 .             .
 .             .
PARM         parmN
```

Programming Language Considerations

This section describes the programming language considerations a programmer should keep in mind when writing and running a program that uses CPI Communications. Programming language considerations that apply only to a certain environment are not listed in this section, but they are included in the appendix that describes that environment.

A pseudonym file or dataset for each supported programming language is available on most implementing operating systems. See the appropriate appendix in this book for the names of the pseudonym files used in your environment. See Appendix L, “Pseudonym Files” on page 335 for sample pseudonym files for the SAA programming languages that support CPI Communications.

Application Generator

Cross System Product (CSP) is the implementing product for the Application Generator common programming interface. No special considerations apply to CSP programs using CPI Communications routines.

C

The following notes apply to C programs using CPI Communications routines:

- When passing an integer value as a parameter, prefix the parameter name with an ampersand (&) so that the value is passed by reference.
- To pass a parameter as a string literal, surround it with double quotes rather than single quotes.

COBOL

The following notes apply to COBOL programs using CPI Communications routines:

- Because COBOL does not support the underscore character (_), the underscores in COBOL pseudonyms are replaced with dashes (-). For example, COBOL programmers 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 full words (at least five but less than ten "9"s) and they must be COMP-4, not zoned decimal.

FORTRAN

The following note applies to FORTRAN programs using CPI Communications routines:

- The EXTERNAL statement may be required for each CPI Communications routine that is called, depending on the environment being used. The PRAGMA statement may also be required. EXTERNAL and PRAGMA statements may be included in the FORTRAN pseudonym file provided for a given environment.

PL/I

The following notes apply to PL/I programs using CPI Communications routines:

- Numbers in the parameter list must be declared, initialized, and passed as variables.
- ENTRY declaration statements should be used for each CPI Communications routine that is called. ENTRY declaration statements may be included in the PL/I pseudonym file provided for a given environment.

Procedures Language

REXX is the implementing product for the Procedures Language common programming interface. The following notes apply to REXX programs using CPI Communications routines:

- REXX programs must use the ADDRESS CPICOMM statement to access CPI Communications routines. These routines are not accessible through the REXX CALL statement interface.

- Character strings returned by CPI Communications routines are stored in variables with the maximum allowable length. (Maximum lengths are shown in Appendix A, "Variables and Characteristics" on page 147.) 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 fixed-length character strings. For instance, a *conversation_ID* returned from the Accept_Conversation call always has a length of 8 bytes.)

RPG

The following note applies to RPG programs using CPI Communications routines:

- Because RPG supports only variable names with lengths of 1 to 6 characters, the pseudonym names for RPG have been abbreviated. For example, RPG programmers should use IMMED as a pseudonym name in their programs instead of CM_IMMEDIATE.

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 137 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 58 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, which goes into the call as a parameter, is a 32-bit integer. This information is provided in Table 7 on page 153.
- Table 5 on page 148 in Appendix A, "Variables and Characteristics" shows that CM_IMMEDIATE, which is placed into the *return_control* parameter on the call to CMSRC, is defined as having an integer value of 1.
- Finally, the *return_code* value CM_OK, which is returned to the program on the CMSRC call, is defined in Appendix B, "Return Codes." CM_OK 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 — begins on page 63. Key-topic discussions and where they occur are:

- "Naming Conventions — Calls and Characteristics, Variables and Values" on page 22 describes the naming conventions used throughout the book.
- "Data Buffering and Transmission" on page 39 provides a discussion of program control over data transmission.
- "Usage Notes" of "Request_To_Send (CMRTS)" on page 105 discusses how a conversation enters **Receive** state.
- "Usage Notes" of "Send_Data (CMSEND)" on page 110 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.	65
CMALLC	Allocate	Used by a program to establish a conversation.	67
CMCFM	Confirm	Used by a program to send a confirmation request to its partner.	70
CMCFMD	Confirmed	Used by a program to send a confirmation reply to its partner.	73
CMDEAL	Deallocate	Used by a program to end a conversation.	75
CMECS	Extract_Conversation_State	Used by a program to view the current state of a conversation.	79
CMECT	Extract_Conversation_Type	Used by a program to view the current *conversation_type* conversation characteristic.	81
CMEMN	Extract_Mode_Name	Used by a program to view the current *mode_name* conversation characteristic.	82
CMEPLN	Extract_Partner_LU_Name	Used by a program to view the current *partner_LU_name* conversation characteristic.	84
CMESL	Extract_Sync_Level	Used by a program to view the current *sync_level* conversation characteristic.	86
CMFLUS	Flush	Used by a program to flush the LU's send buffer.	88
CMINIT	Initialize_Conversation	Used by a program to initialize the conversation characteristics.	90
CMPTR	Prepare_To_Receive	Used by a program to change a conversation from **Send** to **Receive** state in preparation to receive data.	93
CMRCV	Receive	Used by a program to receive data.	97
CMRTS	Request_To_Send	Used by a program to notify its partner that it would like to send data.	105
CMSCT	Set _Conversation_Type	Used by a program to set the *conversation_type* conversation characteristic.	118
CMSDT	Set_Deallocate_Type	Used by a program to set the *deallocate_type* conversation characteristic.	120
CMSED	Set_Error_Direction	Used by a program to set the *error_direction* conversation characteristic.	123
CMSEND	Send_Data	Used by a program to send data.	107

Call	Pseudonym	Description	Page
CMSERR	Send_Error	Used by a program to notify its partner of an error that occurred during the conversation.	112
CMSF	Set_Fill	Used by a program to set the *fill* conversation characteristic.	125
CMSLD	Set_Log_Data	Used by a program to set the *log_data* conversation characteristic.	127
CMSMN	Set_Mode_Name	Used by a program to set the *mode_name* conversation characteristic.	129
CMSPLN	Set_Partner_LU_Name	Used by a program to set the *partner_LU_name* conversation characteristic.	131
CMSPTR	Set_Prepare_To_Receive_Type	Used by a program to set the *prepare_to_receive_type* conversation characteristic.	133
CMSRC	Set_Return_Control	Used by a program to set the *return_control* conversation characteristic.	137
CMSRT	Set_Receive_Type	Used by a program to set the *receive_type* conversation characteristic.	135
CMSSL	Set_Sync_Level	Used by a program to set the *sync_level* conversation characteristic.	141
CMSST	Set_Send_Type	Used by a program to set the *send_type* conversation characteristic.	139
CMSTPN	Set_TP_Name	Used by a program to set the *TP_Name* conversation characteristic.	143
CMTRTS	Test_Request_To_Send_Received	Used by a program to determine whether or not the remote program is requesting to send data.	145

Accept_Conversation (CMACCP)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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.

Format

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

Parameters

conversation_ID (output)
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 (output)
Specifies the result of the call execution. 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 conversation enters the **Receive** state.

Usage Notes

1. 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 program must issue the Accept_Conversation call before any other calls can refer to the conversation.

2. Although CPI Communications allows a program to accept only one conversation, there is no limitation on the number of conversations that a program can allocate. For example, the program might accept one conversation (receiving a unique *conversation_ID*) and then allocate another conversation (receiving another unique *conversation_ID*).

3. For a list of the conversation characteristics that are initialized when the Accept_Conversation call completes successfully, see Table 3 on page 19.

Related Information

"Conversation Characteristics" on page 18 provides a comparison of the conversation characteristics set by Accept_Conversation and Initialize_Conversation.

"Example 1: Data Flow in One Direction" on page 33 shows an example program flow using the Accept_Conversation call.

"Initialize_Conversation (CMINIT)" on page 90 describes how the conversation characteristics are initialized for the program that allocates the conversation.

VM provides extensions to CPI Communications that enable an application to accept more than one concurrent conversation. See Appendix J, "CPI Communications on VM/ESA CMS" on page 279 for more information.

Allocate (CMALLC)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	X

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

Before issuing the Allocate call, a program has the option of issuing one or more of the following calls to set allocation parameters:

CALL CMSCT – Set_Conversation_Type
CALL CMSMN – Set_Mode_Name
CALL CMSPLN – Set_Partner_LU_Name
CALL CMSRC – Set_Return_Control
CALL CMSSL – Set_Sync_Level
CALL CMSTPN – Set_TP_Name

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier of an initialized conversation.

return_code *(output)*
Specifies the result of the call execution. The *return_code* variable can have the following values:

- CM_OK
- CM_SYNC_LVL_NOT_SUPPORTED_LU
- 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 a mode name that is used by SNA service transaction programs only, such as SNASVCMG, but the local program does not have the authority to specify this mode name.
  - 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 one of the following:
  - The conversation is not in **Initialize** state.
  - For a conversation with *sync_level* set to CM_SYNC_POINT, the transaction program is in the **Backout-Required** condition. New protected conversations cannot be allocated when the transaction program is in this condition.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PRODUCT_SPECIFIC_ERROR

In addition, when *return_control* is set to CM_WHEN_SESSION_ALLOCATED, *return_code* can have the following values:

- CM_ALLOCATE_FAILURE_NO_RETRY
- CM_ALLOCATE_FAILURE_RETRY

If *return_control* is set to CM_IMMEDIATE, *return_code* can have the following value:

- CM_UNSUCCESSFUL

  This value indicates that the session is not immediately available.

State Changes

When *return_code* is set to CM_OK, the conversation 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. A set of security parameters is established for the conversation (using a default value of SECURITY=SAME). CPI Communications does not provide a method for the program to modify or examine the security parameters, although such a method may be provided as a CPI Communications extension in some environments. See the appropriate product appendix for information about a particular environment.

6. After making a call to Accept_Conversation, the remote program's end of the conversation is in **Receive** state.

Related Information

"Example 1: Data Flow in One Direction" on page 33 shows an example program flow using the Allocate call.

"SNA Service Transaction Programs" on page 183 provides a discussion of SNA service transaction programs.

"Data Buffering and Transmission" on page 39 provides a complete discussion of control methods for data transmission.

"Set_Return_Control (CMSRC)" on page 137 provides a discussion of the *return_control* characteristic.

"Set_Conversation_Type (CMSCT)" on page 118 provides a discussion of the *conversation_type* characteristic.

Confirm (CMCFM)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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 or CM_SYNC_POINT to use this call. The Set_Sync_Level (CMSSL) call is used to set a conversation's synchronization level.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

request_to_send_received *(output)*
Specifies the variable containing 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 because the remote program issued a Request_To_Send. This is a request that the local program's end of the conversation enter **Receive** state, which will place the remote program's end of the conversation in **Send** state.
- CM_REQ_TO_SEND_NOT_RECEIVED
  A request-to-send notification was not received.

Note: When *return_code* indicates CM_PROGRAM_PARAMETER_CHECK or CM_PROGRAM_STATE_CHECK, the value contained in *request_to_send_received* is meaningless.

return_code *(output)*
Specifies the result of the call execution. The *return_code* variable can have one of the following values:

- CM_OK (remote program replied Confirmed)
- CM_CONVERSATION_TYPE_MISMATCH
- CM_PIP_NOT_SPECIFIED_CORRECTLY
- 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_DEALLOCATED_ABEND_SVC
- CM_DEALLOCATED_ABEND_TIMER
- CM_SVC_ERROR_PURGING
- 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.
  - For a conversation with *sync_level* set to CM_SYNC_POINT, the transaction program is in the **Backout-Required** condition. The Confirm call is not allowed while the transaction program is in this condition.
- 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
- The following values are returned only when *sync_level* is set to CM_SYNC_POINT:
  - CM_TAKE_BACKOUT
  - CM_DEALLOCATED_ABEND_BO
  - CM_DEALLOCATED_ABEND_SVC_BO
  - CM_DEALLOCATED_ABEND_TIMER_BO
  - CM_RESOURCE_FAIL_NO_RETRY_BO
  - CM_RESOURCE_FAILURE_RETRY_BO

State Changes

When *return_code* is set to CM_OK:

- The conversation enters **Send** state if the program issued the Confirm call with the conversation in **Send-Pending** state.
- The conversation enters **Receive** state if the program issued the Confirm call with the conversation in **Defer-Receive** state.
- No state change occurs if the program issued the Confirm call with the conversation in **Send** state.

Usage Notes

1. The program that issues Confirm waits until a reply from the remote partner program is received. (This reply is made using the Confirmed call.)
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 73 provides information on the remote program's reply to the Confirm call.

"Request_To_Send (CMRTS)" on page 105 provides a complete discussion of the *request_to_send_received* parameter.

"Set_Sync_Level (CMSSL)" on page 141 explains how programs specify the level of synchronization processing.

"Example 4: Validation and Confirmation of Data Reception" on page 42 shows an example program using the Confirm call.

Confirmed (CMCFMD)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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 *(input)*
Specifies the conversation identifier.

return_code *(output)*
Specifies the result of the call execution. 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.
  - For a conversation with *sync_level* set to CM_SYNC_POINT, the program may be in the **Backout-Required** condition. The Confirmed call is not allowed while the program is in this condition.
- 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 conversation enters **Receive** state if the program received the *status_received* parameter set to CM_CONFIRM_RECEIVED on the preceding Receive call — that is, if the conversation was in **Confirm** state.
- The conversation enters **Send** state if the program received the *status_received* parameter set to CM_CONFIRM_SEND_RECEIVED on the preceding Receive call — that is, if the conversation was in **Confirm-Send** state.
- The conversation enters **Reset** state if the program received the *status_received* parameter set to CM_CONFIRM_DEALLOC_RECEIVED on the preceding Receive call — that is, if 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 70 provides more information on the Confirm call.

"Receive (CMRCV)" on page 97 provides more information on the *status_received* parameter.

"Set_Sync_Level (CMSSL)" on page 141 explains how programs specify the level of synchronization processing.

"Example 4: Validation and Confirmation of Data Reception" on page 42 shows an example program using the Confirmed call.

Deallocate (CMDEAL)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	X

A program uses the Deallocate (CMDEAL) call to end a conversation. The deallocation can either be completed as part of this call or deferred until the program issues an SAA resource recovery interface Commit Call. When it is completed as part of this call, 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 conversation is deallocated as part of this call.

Before issuing the Deallocate call, a program has the option of issuing one or both of the following calls to set deallocation parameters:

CALL CMSDT – Set_Deallocate_Type
CALL CMSLD – Set_Log_Data

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier of the conversation to be ended.

return_code *(output)*
Specifies the result of the call execution.

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 completed)
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - 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 completed)
- CM_CONVERSATION_TYPE_MISMATCH
- CM_PIP_NOT_SPECIFIED_CORRECTLY
- 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_DEALLOCATED_ABEND_SVC
- CM_DEALLOCATED_ABEND_TIMER
- CM_SVC_ERROR_PURGING
- 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

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

- CM_OK
  Deallocation is deferred until the program issues an SAA resource recovery interface Commit call. If the Commit call is successful, the conversation will be deallocated normally. If the Commit is not successful or if the program issues a resource recovery interface Backout call instead of a Commit, the conversation will not be deallocated. Instead, the conversation will be restored to the state it was in at the previous synchronization point. Table 9 on page 179 shows how the resource recovery interface calls affect CPI Communications conversation states.
- 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.
  - The *sync_level* is set to CM_SYNC_POINT, and the transaction program is in the **Backout-Required** condition.

- 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 conversation enters **Reset** state if *sync_level* is not set to CM_SYNC_POINT.
- The conversation enters **Defer-Deallocate** state if *sync_level* is set to CM_SYNC_POINT and *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL.
- The conversation enters **Reset** state if *sync_level* is set to CM_SYNC_POINT and *deallocate_type* is set to CM_DEALLOCATE_ABEND.

Usage Notes

1. If *deallocate_type* is set to CM_DEALLOCATE_FLUSH, CM_DEALLOCATE_CONFIRM, or CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is CM_NONE or CM_CONFIRM, the execution of Deallocate includes the flushing of the local LU's send buffer. If *deallocate_type* is CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is CM_SYNC_POINT, the local LU's send buffer will not be flushed until an SAA resource recovery interface Commit or Backout call is issued by the TP or the sync point manager.
2. If a conversation is using *sync_level*=CM_SYNC_POINT, CPI Communications does not allow the conversation to be deallocated with a *deallocate_type* of CM_DEALLOCATE_CONFIRM or CM_DEALLOCATE_FLUSH.
3. 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 the appropriate format. After completion of the Deallocate processing, the *log_data* is reset to null and the *log_data_length* is reset to zero.
4. 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 either with the *deallocate_type* set to CM_DEALLOCATE_FLUSH or with the *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL and the *sync_level* characteristic set to CM_NONE.
   - 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 with its end of the conversation 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 112 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.
   - CM_TAKE_COMMIT_DEALLOCATE *status_received* indication
     This indicates that the partner program issued an SAA resource recovery interface Commit call after issuing a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT.

5. The program should terminate all conversations before the end of the program. However, if the program does not terminate all conversations, node services will abnormally deallocate any dangling conversations. See the appropriate appendix in this book for a description of how dangling conversations are deallocated in a specific environment.

Related Information

"Example 1: Data Flow in One Direction" on page 33 shows an example program flow using the Deallocate call.

"Set_Log_Data (CMSLD)" on page 127 provides a discussion of the *log_data* characteristic.

"Set_Deallocate_Type (CMSDT)" on page 120 provides a discussion of the *deallocate_type* characteristic and its possible values.

Extract_Conversation_State (CMECS)

MVS	VM	OS/400	OS/2	IMS	CICS
	X		X		

A program uses the Extract_Conversation_State (CMECS) call to extract the conversation state for a given conversation. The value is returned in the *conversation_state* parameter.

Format

```
CALL CMECS(conversation_ID,
           conversation_state,
           return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

conversation_state *(output)*
Specifies the conversation state that is returned to the local program. The *conversation_state* can be one of the following:

- CM_INITIALIZE_STATE
- CM_SEND_STATE
- CM_RECEIVE_STATE
- CM_SEND_PENDING_STATE
- CM_CONFIRM_STATE
- CM_CONFIRM_SEND_STATE
- CM_CONFIRM_DEALLOCATE_STATE
- CM_DEFER_RECEIVE_STATE
- CM_DEFER_DEALLOCATE_STATE
- CM_SYNC_POINT_STATE
- CM_SYNC_POINT_SEND_STATE
- CM_SYNC_POINT_DEALLOCATE_STATE

Notes:

1. Unless *return_code* is set to CM_OK, the value of *conversation_state* is not meaningful.

2. The following *conversation_state* values are not returned on OS/2 systems, because OS/2 does not support a *sync_level* of CM_SYNC_POINT:
   - CM_DEFER_RECEIVE_STATE
   - CM_DEFER_DEALLOCATE_STATE
   - CM_SYNC_POINT_STATE
   - CM_SYNC_POINT_SEND_STATE
   - CM_SYNC_POINT_DEALLOCATE_STATE

return_code *(output)*

Specifies the result of the call execution. 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_TAKE_BACKOUT
  This value is returned only when all of the following conditions are true:
  - The *sync_level* is set to CM_SYNC_POINT.
  - The conversation is not in Initialize state.
  - The application is in the **Backout-Required** condition.
  - The program is using protected resources that must be backed out.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

This call can be used to discover the state of a conversation after it has been backed out during an SAA resource recovery interface Backout operation.

Related Information

"Support for the SAA Resource Recovery Interface" on page 25 provides more information on the SAA resource recovery interface.

Extract_Conversation_Type (CMECT)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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 *(input)*
Specifies the conversation identifier.

conversation_type *(output)*
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: Unless *return_code* is set to CM_OK, the value of *conversation_type* is not meaningful.

return_code *(output)*
Specifies the result of the call execution. 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 118 provides more information on the *conversation_type* characteristic.

Extract_Mode_Name (CMEMN)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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 *(input)*
Specifies the conversation identifier.

mode_name *(output)*
Specifies the variable containing the mode name. 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: Unless *return_code* is set to CM_OK, the value of *mode_name* is not meaningful.

mode_name_length *(output)*
Specifies the variable containing the length of the returned *mode_name* parameter.

Note: Unless *return_code* is set to CM_OK, the value of *mode_name_length* is not meaningful.

return_code *(output)*
Specifies the result of the call execution. 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

1. This call does not change the *mode_name* for the specified conversation.
2. CPI Communications returns the *mode_name* using the native encoding of the local system.

Related Information

"Set_Mode_Name (CMSMN)" on page 129 and "Side Information" on page 15 provide further information on the *mode_name* characteristic.

"Automatic Conversion of Characteristics" on page 20 provides further information on the automatic conversion of the *mode_name* parameter.

Extract_Partner_LU_Name (CMEPLN)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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 *(input)*
Specifies the conversation identifier.

partner_LU_name *(output)*
Specifies the variable containing the name of the LU where the remote program is located.

Note: Unless *return_code* is set to CM_OK, the value of *partner_LU_name* is not meaningful.

partner_LU_name_length *(output)*
Specifies the variable containing the length of the returned *partner_LU_name* parameter.

Note: Unless *return_code* is set to CM_OK, the value of *partner_LU_name_length* is not meaningful.

return_code *(output)*
Specifies the result of the call execution. 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

1. This call does not change the *partner_LU_name* for the specified conversation.
2. CPI Communications returns the *partner_LU_name* using the native encoding of the local system.

Related Information

"Set_Partner_LU_Name (CMSPLN)" on page 131 and "Side Information" on page 15 provide more information on the *partner_LU_name* characteristic.

"Automatic Conversion of Characteristics" on page 20 provides further information on the automatic conversion of the *partner_LU_name* parameter.

Extract_Sync_Level (CMESL)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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 *(input)*
Specifies the conversation identifier.

sync_level *(output)*
Specifies the variable containing the *sync_level* characteristic of this conversation. 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 calls or recognize 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.
- CM_SYNC_POINT
  For systems that support SAA resource recovery interface processing, this value specifies that this conversation is a protected resource. The programs can issue resource recovery interface calls and will recognize returned parameters relating to resource recovery interface operations. The programs can also perform confirmation processing.

  The CM_SYNC_POINT value is not returned on CICS, IMS, MVS, OS/2, or OS/400 systems.

Note: Unless *return_code* is set to CM_OK, the value of *sync_level* is not meaningful.

return_code *(output)*
Specifies the result of the call execution. 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 141 provides more information on the *sync_level* characteristic.

Flush (CMFLUS)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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 call. 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 *(input)*
Specifies the conversation identifier.

return_code *(output)*
Specifies the result of the call execution. 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**, **Send-Pending**, or **Defer-Receive** state.
  - For a conversation with *sync_level* set to CM_SYNC_POINT, the program may be in the **Backout-Required** condition. The Flush call is not allowed while the program is in this condition.
- 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 conversation enters **Send** state if the program issues the Flush call with the conversation in **Send-Pending** state.
- The conversation enters **Receive** state if the program issues the Flush call with the conversation in **Defer-Receive** state.
- No state change occurs if the program issues the Flush call with the conversation 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.

   The local program can issue a Flush call to cause the LU to transmit the data immediately. This helps minimize any delay in the remote program's processing of the data.

2. The Flush call causes the local LU to flush its send buffer only when the LU 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 139 provides a discussion of alternative methods of achieving the Flush function.

"Allocate (CMALLC)" on page 67 provides more information on how information is buffered from the Allocate call.

"Send_Data (CMSEND)" on page 107 provides more information on how information is buffered from the Send_Data call.

"Send_Error (CMSERR)" on page 112 provides more information on how information is buffered from the Send_Error call.

"Data Buffering and Transmission" on page 39 provides a complete discussion of the conditions for data transmission.

"Example 4: Validation and Confirmation of Data Reception" on page 42 shows an example of how a program can use the Flush call to establish a conversation immediately.

Initialize_Conversation (CMINIT)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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.

Format

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

Parameters

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

sym_dest_name *(input)*
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. Alternatively, a blank *sym_dest_name* (one composed of eight space characters) may be specified. When this is done, the program is responsible for setting up the appropriate destination information, using Set calls, before issuing the Allocate call for that conversation.

IMS and MVS do not support blank *sym_dest_name* values on the Initialize_Conversation call.

On VM, if no corresponding entry is found in the side information table, the name provided in *sym_dest_name* will be used as the partner TP name.

return_code *(output)*
Specifies the result of the call execution. The *return_code* variable can have one of the following values:

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

VM does not return CM_PROGRAM_PARAMETER_CHECK when an unrecognized *sym_dest_name* is specified. Instead, CM_OK is returned and the unrecognized *sym_dest_name* is assumed to be the name of the partner transaction program. See Appendix J, "CPI Communications on VM/ESA CMS" on page 279 for more information.

State Changes

When *return_code* indicates CM_OK, the conversation enters the **Initialize** state.

Usage Notes

1. For a list of the conversation characteristics that are initialized when the Initialize_Conversation call completes successfully, see Table 3 on page 19.
2. 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.
3. A program can call Initialize_Conversation more than once and establish multiple, concurrently active conversations. When a program with an existing initialized conversation issues an Initialize_Conversation call, CPI Communications initializes a new conversation and assigns a new *conversation_ID*. CPI Communications is designed so that Initialize_Conversation is always issued from the **Reset** state. For more information about managing concurrent conversations, see "Multiple Conversations" on page 24.
4. If the side information table supplies invalid allocation information on the Initialize_Conversation (CMINIT) call, or if the program supplies invalid allocation information on any subsequent Set calls, the error is detected when the information is processed by Allocate (CMALLC).
5. A set of security parameters is established for the conversation (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, although such a method may be provided as a CPI Communications extension in some environments. See the appropriate product appendix for information about a particular environment.
6. A program may obtain information about its partner program (that is, *partner_LU_name*, *TP_name*, and *mode_name*) from a source other than the side information table. The local program can, for example, read this information from a file or receive it from another partner over a separate conversation. The information might even be hard-coded in the program. In cases where a program wishes to specify destination information about its partner program without making use of side information, the local program may supply a blank *sym_dest_name* on the Initialize_Conversation call. CPI Communications will initialize the conversation characteristics and return a *conversation_ID* for the new conversation. The program is then responsible for specifying valid destination information (via Set_Partner_LU_Name, Set_TP_Name and Set_Mode_Name calls) before issuing the Allocate call.

Related Information

"Accept_Conversation (CMACCP)" on page 65 provides more information on how conversation characteristics are set by the Accept_Conversation call.

"Allocate (CMALLC)" on page 67 provides more information on how conversation characteristics are set by the Allocate call.

"Conversation Characteristics" on page 18 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 33 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 15 provides more information on *sym_dest_name*.

Prepare_To_Receive (CMPTR)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	X

A program uses the Prepare_To_Receive (CMPTR) call to change a conversation from **Send** to **Receive** state in preparation to receive data. The change to **Receive** state can be either completed as part of this call or deferred until the program issues a Flush, Confirm, or SAA resource recovery interface Commit call. When the change to **Receive** state is completed as part of this call, it may include the function of the Flush or Confirm call. This call's function is determined by the value of the *prepare_to_receive_type* conversation characteristic.

Before issuing the Prepare_To_Receive call, a program has the option of issuing the following call which affects the function of the Prepare_To_Receive call:

Call CMSPTR — Set_Prepare_To_Receive_Type

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

return_code *(output)*
Specifies the result of the call execution. 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.
  - The *sync_level* is set to CM_SYNC_POINT, and the transaction program is in the **Backout-Required** condition. The Prepare_To_Receive call is not allowed while the transaction program is in this condition.
- 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_PIP_NOT_SPECIFIED_CORRECTLY
- 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_DEALLOCATED_ABEND_SVC
- CM_DEALLOCATED_ABEND_TIMER
- CM_SVC_ERROR_PURGING
- 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.
  - For a conversation with *sync_level* set to CM_SYNC_POINT, the transaction program is in the **Backout-Required** condition. The Prepare_To_Receive call is not allowed while the transaction program is in this condition.
- CM_PROGRAM_PARAMETER_CHECK
  This return code indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PRODUCT_SPECIFIC_ERROR
- The following values are returned only when *sync_level* is set to CM_SYNC_POINT:
  - CM_TAKE_BACKOUT
  - CM_DEALLOCATED_ABEND_BO
  - CM_DEALLOCATED_ABEND_SVC_BO
  - CM_DEALLOCATED_ABEND_TIMER_BO
  - CM_RESOURCE_FAIL_NO_RETRY_BO
  - CM_RESOURCE_FAILURE_RETRY_BO

If *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and the *sync_level* for the conversation is CM_SYNC_POINT, *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.
  - The transaction program is in the **Backout-Required** condition.
- CM_PROGRAM_PARAMETER_CHECK
  This return code indicates that the *conversation_ID* specifies an unassigned conversation identifier.

State Changes

When *return_code* indicates CM_OK:

- If any of the following conditions is true, the conversation enters the **Receive** state.
  - The *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_CONFIRM
  - The *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* is set to CM_CONFIRM
  - The *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_FLUSH
  - The *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* is set to CM_NONE
- The conversation enters the **Defer-Receive** state if *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* is set to CM_SYNC_POINT.

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, the LU's send buffer is flushed if it contains information, and send control is transferred to the remote program without any synchronizing acknowledgment.

   - If *sync_level* is set to CM_CONFIRM, the LU's send buffer is flushed if it contains information, and send control is transferred to the remote program with confirmation requested.

   - If *sync_level* is set to CM_SYNC_POINT, transfer of send control is deferred. When the local program subsequently issues a Flush, Confirm, or SAA resource recovery interface Commit call, the LU's send buffer is flushed if it contains information, and send control is transferred to the remote program. (A synchronization point is also requested when the call is a Commit call).

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.
   - CM_TAKE_COMMIT_SEND
     The local program issued an SAA resource recovery interface Commit call after issuing the Prepare_To_Receive call with *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT.

6. When the local program's end of the conversation enters **Receive** state, the remote program's end of the conversation enters **Send** or **Send-Pending** state, depending on the *data_received* indicator. The remote program can then send data to the local program.

7. If the local program issued the call with *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT, the LU buffers the send notification to be sent to the remote partner until the local program issues a call that causes the LU to flush its send buffer.

Related Information

"Set_Prepare_To_Receive_Type (CMSPTR)" on page 133 provides more information on the *prepare_to_receive_type* characteristic.

"Set_Sync_Level (CMSSL)" on page 141 provides a discussion of the *sync_level* characteristic and its possible values.

"Example 3: The Sending Program Changes the Data Flow Direction" on page 40 and "Example 5: The Receiving Program Changes the Data Flow Direction" on page 44 show example program flows where the Prepare_To_Receive call is used.

Receive (CMRCV)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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 or for SAA resource recovery interface services.

Before issuing the Receive call, a program has the option of issuing one or both of the following calls, which affect the function of the Receive call:

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 *(input)*
Specifies the conversation identifier.

buffer *(output)*
Specifies the variable in which the program is to receive the data.

Note: *Buffer* contains data only if *return_code* is set to CM_OK or CM_DEALLOCATED_NORMAL and *data_received* is not set to CM_NO_DATA_RECEIVED.

requested_length *(input)*
Specifies the maximum amount of data the program is to receive. Valid *requested_length* values range from 0 to 32767.

data_received ***(output)***
Specifies whether or not the program received data.

Note: Unless *return_code* is set to CM_OK or CM_DEALLOCATED_NORMAL, the value contained in *data_received* is meaningless.

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 *(output)*
Specifies the variable containing the amount of data the program received, up to the maximum. If the program receives information other than data, the value contained in *received_length* is meaningless.

status_received *(output)*
Specifies the variable containing an indication of the conversation status.

Note: Unless *return_code* is set to CM_OK, the value contained in *status_received* is meaningless.

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's end of the conversation has entered **Receive** state, placing the local program's end of the conversation 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 (which issued the Receive call) can now 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's end of the conversation 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. Upon issuing a successful Confirmed call, the local program (which issued the Receive call) can now send data.
- 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. Upon issuing a successful Confirmed call, the local program (which issued the Receive call) is deallocated—that is, placed in **Reset** state.

For a conversation with *sync_level* set to CM_SYNC_POINT, the *status_received* variable can also be set to one of the following values:

- CM_TAKE_COMMIT
  The remote program has issued an SAA resource recovery interface Commit call. The local program should issue a Commit call in order to commit all protected resources throughout the transaction. When appropriate, the local program may respond by issuing a call other than Commit, such as Send_Error or an SAA resource recovery interface Backout call.
- CM_TAKE_COMMIT_SEND
  The remote program has issued a Prepare_To_Receive call with *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT and then issued a resource recovery interface Commit call. The local program should issue a Commit call in order to commit all protected resources throughout the transaction. When appropriate, the local program may respond by issuing a call other than Commit, such as Send_Error or a resource recovery interface Backout call. If a successful Commit call is issued, the local program can then send data.
- CM_TAKE_COMMIT_DEALLOCATE
  The remote program has deallocated the conversation with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT and then issued a resource recovery interface Commit call. The local program should issue a Commit call in order to commit all protected resources throughout the transaction. The local program may respond by issuing a call other than Commit when appropriate, such as Send_Error or a resource recovery interface Backout call. If a successful Commit call is issued, the local program is then deallocated—that is, placed in **Reset** state.

request_to_send_received *(output)*
Specifies the variable containing 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, the value contained in *request_to_send_received* is meaningless.

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's end of the conversation to enter **Receive** state, which would place the remote program's end of the conversation in **Send** state. See "Request_To_Send (CMRTS)" on page 105 for further discussion of the local program's possible responses.
- CM_REQ_TO_SEND_NOT_RECEIVED
  The local program has not received a request-to-send notification.

return_code *(output)*
Specifies the result of the call execution. 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_PIP_NOT_SPECIFIED_CORRECTLY

- 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_DEALLOCATED_ABEND_SVC
- CM_DEALLOCATED_ABEND_TIMER
- CM_SVC_ERROR_NO_TRUNC
- CM_SVC_ERROR_PURGING
- CM_PRODUCT_SPECIFIC_ERROR
- The following values are returned only when *sync_level* is set to CM_SYNC_POINT:
  - CM_TAKE_BACKOUT
  - CM_DEALLOCATED_ABEND_BO
  - CM_DEALLOCATED_ABEND_SVC_BO
  - CM_DEALLOCATED_ABEND_TIMER_BO
  - CM_RESOURCE_FAIL_NO_RETRY_BO
  - CM_RESOURCE_FAILURE_RETRY_BO

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_DEALLOCATED_ABEND_SVC
- CM_DEALLOCATED_ABEND_TIMER
- CM_SVC_ERROR_NO_TRUNC
- CM_SVC_ERROR_PURGING
- CM_PRODUCT_SPECIFIC_ERROR
- The following values are returned only when *sync_level* is set to CM_SYNC_POINT:
  - CM_TAKE_BACKOUT
  - CM_DEALLOCATED_ABEND_BO
  - CM_DEALLOCATED_ABEND_SVC_BO
  - CM_DEALLOCATED_ABEND_TIMER_BO
  - CM_RESOURCE_FAIL_NO_RETRY_BO
  - CM_RESOURCE_FAILURE_RETRY_BO

If *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_PIP_NOT_SPECIFIED_CORRECTLY
- 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_DEALLOCATED_ABEND_SVC
- CM_DEALLOCATED_ABEND_TIMER
- CM_SVC_ERROR_NO_TRUNC
- CM_SVC_ERROR_PURGING
- CM_SVC_ERROR_TRUNC (basic conversation only)
- CM_UNSUCCESSFUL
  This value indicates that *receive_type* is set to CM_RECEIVE_IMMEDIATE, but there is nothing to receive.
- CM_PRODUCT_SPECIFIC_ERROR
- The following values are returned only when *sync_level* is set to CM_SYNC_POINT:
  - CM_TAKE_BACKOUT
  - CM_DEALLOCATED_ABEND_BO
  - CM_DEALLOCATED_ABEND_SVC_BO
  - CM_DEALLOCATED_ABEND_TIMER_BO
  - CM_RESOURCE_FAIL_NO_RETRY_BO
  - CM_RESOURCE_FAILURE_RETRY_BO

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 in **Send** state; and the program started but did not finish sending a logical record.
  - The *sync_level* is set to CM_SYNC_POINT, and the transaction program is in the **Backout-Required** condition. The Receive call is not allowed while the transaction program is in this condition.
- 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 conversation 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 conversation enters **Send** state when *data_received* is set to CM_NO_DATA_RECEIVED and *status_received* is set to CM_SEND_RECEIVED.
- The conversation enters **Send-Pending** state when *data_received* is set to CM_DATA_RECEIVED, CM_COMPLETE_DATA_RECEIVED, or CM_INCOMPLETE_DATA_RECEIVED, and *status_received* is set to CM_SEND_RECEIVED.
- The conversation 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.
- For a conversation with *sync_level* set to CM_SYNC_POINT, the conversation enters **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** state when *status_received* is set to CM_TAKE_COMMIT, CM_TAKE_COMMIT_SEND, or CM_TAKE_COMMIT_DEALLOCATE, respectively.
- 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 a Receive call with its end of the conversation 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's end of the conversation 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 93 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.

   Since data may not be available when a given Receive call is issued, a program that is using concurrent conversations with multiple partners might use a *receive_type* of CM_RECEIVE_IMMEDIATE and periodically check each conversation for data. For more information about multiple, concurrent conversations, see "Multiple Conversations" on page 24.

4. If the *return_code* indicates CM_PROGRAM_STATE_CHECK or CM_PROGRAM_PARAMETER_CHECK, the values of all other parameters on this call are meaningless.

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 intends 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 independently 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 under one of the following conditions:

   - *receive_type* is set to CM_RECEIVE_AND_WAIT and the end of the data is received. 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**, **Sync-Point**, **Sync-Point-Send**, **Sync-Point-Deallocate**, or **Reset** state).
   - *receive_type* is set to CM_RECEIVE_IMMEDIATE and an amount of data equal to the *requested_length* specified has not arrived from the partner program.

   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, the conversation is basic, and *fill* is set to CM_FILL_LL, the *data_received* parameter indicates CM_INCOMPLETE_DATA_RECEIVED. If data is available, the conversation is basic, 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 CM_INCOMPLETE_DATA_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 returned to the program in addition to (not in place of) the information indicated by the *return_code*, *data_received*, and *status_received* parameters.

Related Information

"Send_Data (CMSEND)" on page 107 provides more information on complete and incomplete logical records and data records.

"Program Partners and Conversations" on page 13 and "Set_Fill (CMSF)" on page 125 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 105 provides a discussion of how a program can place its end of the conversation into **Receive** state.

"Set_Receive_Type (CMSRT)" on page 135 provides a discussion of the *receive_type* characteristic and its various values.

Request_To_Send (CMRTS)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

return_code *(output)*
Specifies the result of the call execution. 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**, **Confirm-Deallocate**, **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** state.
  - For a conversation with *sync_level* set to CM_SYNC_POINT, the program may be in the **Backout-Required** condition. The Request_To_Send call is not allowed while the program is in this condition.
- 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's end of the conversation to enter **Receive** state in order to place the partner program's end of the conversation (the program that issued the Request_To_Send) in **Send** state.

   The remote program's end of the conversation enters **Receive** state when the remote program successfully issues one of the following calls or sequences of calls:

   - The Receive call with *receive_type* set to CM_RECEIVE_AND_WAIT
   - The Prepare_To_Receive call with *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_FLUSH, CM_PREP_TO_RECEIVE_CONFIRM, or CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_CONFIRM or CM_NONE

- The Send_Data call with *send_type* set to CM_SEND_AND_PREP_TO_RECEIVE and *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_FLUSH, CM_PREP_TO_RECEIVE_CONFIRM, or CM_PREP_TO_RECEIVE_SYNC_LEVEL, and *sync_level* set to CM_CONFIRM or CM_NONE
- The Prepare_To_Receive call with *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT, followed by a successful Commit, Confirm, or Flush call.
- The Send_Data call with *send_type* set to CM_SEND_AND_PREP_TO_RECEIVE, *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, and *sync_level* set to CM_SYNC_POINT, followed by a successful Commit, Confirm, or Flush call.

After a remote program issues one of these calls, the local program's end of the conversation is placed into a corresponding **Send**, **Send-Pending**, **Confirm-Send**, or **Sync-Point-Send** state when the local program issues a Receive call. See the *status_received* parameter for the Receive call on page 97 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's end of the conversation 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 97) 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 97 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 44 shows an example program flow using the Request_To_Send call.

Send_Data (CMSEND)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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.

Before issuing the Send_Data call, a program has the option of issuing one or more of the following calls, which affect the function of the Send_Data call:

CALL CMSST – Set_Send_Type

If *send_type* = CM_SEND_AND_PREP_TO_RECEIVE, optional setup may include:

CALL CMSPTR – Set_Prepare_To_Receive_Type

If *send_type* = CM_SEND_AND_DEALLOCATE, optional setup may include:

CALL CMSDT – Set_Deallocate_Type

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier of the conversation.

buffer *(input)*
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 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.
- A 15-bit binary length of the record

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 *(input)*
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 setup characteristics are significant and retain their meaning as described.

request_to_send_received *(output)*
Specifies the variable containing 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's end of the conversation to enter **Receive** state, which places the remote program's end of the conversation 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, the value contained in *request_to_send_received* is meaningless.

return_code *(output)*
Specifies the result of the call execution. The *return_code* variable can have one of the following values:

- CM_OK
- CM_CONVERSATION_TYPE_MISMATCH
- CM_PIP_NOT_SPECIFIED_CORRECTLY
- 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_DEALLOCATED_ABEND_SVC
- CM_DEALLOCATED_ABEND_TIMER
- CM_SVC_ERROR_PURGING

- 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.
  - The *sync_level* is set to CM_SYNC_POINT, and the transaction program is in the **Backout-Required** condition. The Send_Data call is not allowed while the transaction program is in this condition.
- 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
- The following values are returned only when *sync_level* is set to CM_SYNC_POINT:
  - CM_TAKE_BACKOUT
  - CM_DEALLOCATED_ABEND_BO
  - CM_DEALLOCATED_ABEND_SVC_BO
  - CM_DEALLOCATED_ABEND_TIMER_BO
  - CM_RESOURCE_FAIL_NO_RETRY_BO
  - CM_RESOURCE_FAILURE_RETRY_BO

State Changes

When *return_code* indicates CM_OK:

- The conversation enters **Receive** state when Send_Data is issued with *send_type* set to CM_SEND_AND_PREP_TO_RECEIVE and any of the following conditions are true:

  - *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
  - *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.

- The conversation enters **Defer-Receive** state when Send_Data is issued with *send_type* set to CM_SEND_AND_PREP_TO_RECEIVE, *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, and *sync_level* set to CM_SYNC_POINT.

- The conversation enters **Reset** state when Send_Data is issued with *send_type* set to CM_SEND_AND_DEALLOCATE and any of the following conditions is true:

  - *Deallocate_type* is set to CM_DEALLOCATE_ABEND
  - *Deallocate_type* is set to CM_DEALLOCATE_CONFIRM
  - *Deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is set to CM_CONFIRM
  - *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.

- The conversation enters **Defer-Deallocate** state when Send_Data is issued with *send_type* set to CM_SEND_AND_DEALLOCATE, *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL, and *sync_level* set to CM_SYNC_POINT.
- The conversation 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's end of the conversation to enter **Receive** state, which places the remote program's end of the conversation in **Send** state. See "Request_To_Send (CMRTS)" on page 105 for a discussion of how a program can place its end of a conversation 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 and 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_Data) 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
   - SAA resource recovery interface Commit.

   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 97 provides more information on the *data_received* parameter.

"Program Partners and Conversations" on page 13 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 39 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 139 provides more information on the *send_type* conversation characteristic and the use of it in combination with calls to Send_Data.

Send_Error (CMSERR)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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's end of the conversation is in **Send** state and the remote program's end of the conversation is in **Receive** state. Further action is defined by program logic.

Before issuing the Send_Error call, a program has the option of issuing one or more of the following calls, which affect the function of the Send_Error call:

CALL CMSED – Set_Error_Direction
CALL CMSLD – Set_Log_Data

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

request_to_send_received *(output)*
Specifies the variable containing 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's end of the conversation to enter **Receive** state, which places the remote program's end of the conversation 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, the value contained in *request_to_send_received* is meaningless.

return_code *(output)*
Specifies the result of the call execution. 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_PIP_NOT_SPECIFIED_CORRECTLY
- 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_DEALLOCATED_ABEND_SVC
- CM_DEALLOCATED_ABEND_TIMER
- CM_SVC_ERROR_PURGING
- CM_PRODUCT_SPECIFIC_ERROR
- The following values are returned only when *sync_level* is set to CM_SYNC_POINT:
  - CM_TAKE_BACKOUT
  - CM_DEALLOCATED_ABEND_BO
  - CM_DEALLOCATED_ABEND_SVC_BO
  - CM_DEALLOCATED_ABEND_TIMER_BO
  - CM_PROGRAM_STATE_CHECK
    This return code indicates that the transaction program is in the **Backout-Required** condition. The Send_Error call is not allowed while the transaction program is in this condition.
  - CM_RESOURCE_FAIL_NO_RETRY_BO
  - CM_RESOURCE_FAILURE_RETRY_BO

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
- The following values are returned only when *sync_level* is set to CM_SYNC_POINT:
  - CM_DEALLOCATED_NORMAL_BO
  - CM_PROGRAM_STATE_CHECK
    This return code indicates that the transaction program is in the **Backout-Required** condition. The Send_Error call is not allowed while the transaction program is in this condition.
  - CM_RESOURCE_FAIL_NO_RETRY_BO
  - CM_RESOURCE_FAILURE_RETRY_BO

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

- CM_OK
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_DEALLOCATED_ABEND
- CM_PROGRAM_ERROR_PURGING

- CM_DEALLOCATED_ABEND_SVC
- CM_DEALLOCATED_ABEND_TIMER
- CM_SVC_ERROR_PURGING
- CM_PRODUCT_SPECIFIC_ERROR
- These values are returned only when *sync_level* is set to CM_SYNC_POINT:
  - CM_TAKE_BACKOUT
  - CM_DEALLOCATED_ABEND_BO
  - CM_DEALLOCATED_ABEND_SVC_BO
  - CM_DEALLOCATED_ABEND_TIMER_BO
  - CM_PROGRAM_STATE_CHECK
    This return code indicates that the transaction program is in the **Backout-Required** condition. The Send_Error call is not allowed while the transaction program is in this condition.
  - CM_RESOURCE_FAIL_NO_RETRY_BO
  - CM_RESOURCE_FAILURE_RETRY_BO

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

- CM_OK
- CM_PRODUCT_SPECIFIC_ERROR
- CM_PROGRAM_STATE_CHECK
  This value is returned only when *sync_level* is set to CM_SYNC_POINT. It indicates that the transaction program is in the **Backout-Required** condition. The Send_Error call is not allowed while the transaction program is in this condition.
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_RESOURCE_FAIL_NO_RETRY_BO
  This value is returned only when *sync_level* is set to CM_SYNC_POINT.
- CM_RESOURCE_FAILURE_RETRY_BO
  This value is returned only when *sync_level* is set to CM_SYNC_POINT.

Otherwise, the conversation is in **Reset**, **Initialize**, **Defer-Receive**, or **Defer-Deallocate** 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 conversation enters **Send** state when the call is issued in **Receive**, **Confirm**, **Confirm-Send**, **Confirm-Deallocate**, or **Send-Pending** state. For a conversation with *sync_level* set to CM_SYNC_POINT, the conversation also enters **Send** state when the call is issued in **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** 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 with its end of the conversation in **Send** state after sending an incomplete logical record (see "Send_Data (CMSEND)" on page 107). The record has been truncated.

   - CM_PROGRAM_ERROR_NO_TRUNC (basic and mapped conversations)
     The local program issued Send_Error with its end of the conversation 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 with its end of the conversation 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 with its end of the conversation 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 with its end of the conversation in **Send-Pending** state and *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_CONVERSATION_TYPE_MISMATCH
   - CM_PIP_NOT_SPECIFIED_CORRECTLY
   - 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_ABEND_SVC
   - CM_DEALLOCATED_ABEND_TIMER

Likewise, for conversations with *sync_level* set to CM_SYNC_POINT, a return code of CM_DEALLOCATED_NORMAL_BO is reported instead of:

- CM_DEALLOCATED_ABEND_BO
- CM_DEALLOCATED_ABEND_SVC_BO
- CM_DEALLOCATED_ABEND_TIMER_BO

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)
- CM_SVC_ERROR_NO_TRUNC
- CM_SVC_ERROR_PURGING
- CM_SVC_ERROR_TRUNC (basic conversation only)
- CM_TAKE_BACKOUT

When the return code CM_TAKE_BACKOUT is purged, the remote LU resends the backout indication and the local program receives the CM_TAKE_BACKOUT return code on a subsequent call.

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 call.
  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.
- SAA resource recovery interface Commit call.
  If the Commit call was sent in conjunction with a Deallocate call with *deallocate_type* set to 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 the appropriate format. The data supplied by the program is any data the program wants to have logged. The data is logged on the local system's error log and is also sent to the remote system for logging there. See the appropriate product appendix for more information on error logs.

   After completion of the Send_Error processing, *log_data* is reset to null, and *log_data_length* is reset to zero.

IMS and MVS systems do not send *log_data* to the partner program and do not log data associated with outgoing and incoming Send_Error and Deallocate calls.

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 for a conversation in **Send-Pending** state.

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

Related Information

"Example 6: Reporting Errors" on page 46 and "Example 7: Error Direction and Send-Pending State" on page 48 provide example program flows using Send_Error and the **Send-Pending** state; "Set_Error_Direction (CMSED)" on page 123 provides further information on the *error_direction* characteristic.

"Usage Notes" of "Request_To_Send (CMRTS)" on page 105 provides more information on how a conversation enters **Receive** state.

"Send_Data (CMSEND)" on page 107 provides a discussion of basic conversations and logical records.

"Set_Log_Data (CMSLD)" on page 127 provides a description of the *log_data* characteristic.

Set_Conversation_Type (CMSCT)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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 *(input)*
Specifies the conversation identifier.

conversation_type *(input)*
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 *(output)*
Specifies the result of the call execution. 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. Because of the detailed manipulation of the data and resulting complexity of error conditions, the use of basic conversations should be regarded as an advanced programming technique.
2. 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 13 and the "Usage Notes" section of "Send_Data (CMSEND)" on page 107 provide more information on the differences between mapped and basic conversations.

Set_Deallocate_Type (CMSDT)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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 *(input)*
Specifies the conversation identifier.

deallocate_type *(input)*
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. If the Confirm call is 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.
  - If *sync_level* is set to CM_SYNC_POINT, defer the deallocation until the program issues an SAA resource recovery interface Commit call. If the Commit call is successful, the conversation is deallocated normally. If the Commit is not successful or if the program issues a resource recovery interface Backout call instead of a Commit, the conversation is not deallocated. See "Deallocate (CMDEAL)" on page 75 for more information about deallocating conversations with *sync_level* set to CM_SYNC_POINT.
- 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. If the Confirm call is 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_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 *(output)*
Specifies the result of the call execution. 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 or CM_SYNC_POINT.
  - The *deallocate_type* is set to CM_DEALLOCATE_FLUSH, and the conversation is assigned with *sync_level* set to CM_SYNC_POINT.
  - 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.

   - If *sync_level* is set to CM_SYNC_POINT, the deallocation is deferred until the program issues an SAA resource recovery interface Commit call. If the Commit call is successful, the conversation is deallocated normally. If the Commit is not successful or if the program issues a resource recovery interface Backout call instead of a Commit, the conversation is not deallocated.

2. A *deallocate_type* set to CM_DEALLOCATE_FLUSH is used by a program to unconditionally deallocate the conversation. This *deallocate_type* value can be used for conversations with *sync_level* set to CM_NONE or CM_CONFIRM. 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 75 provides further discussion on the use of the *deallocate_type* characteristic in the deallocation of a conversation.

"Set_Sync_Level (CMSSL)" on page 141 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)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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 call was issued.

Format

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

Parameters

conversation_ID ***(input)***
Specifies the conversation identifier.

error_direction ***(input)***
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 ***(output)***
Specifies the result of the call execution. 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* conversation characteristic 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). 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 for a conversation in **Send-Pending** state.

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

2. If the conversation is in **Send-Pending** state and the program issues a Send_Error call, 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 an error at the remote program occurred in the data before the remote program received send control.

   - 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 an error at the remote program occurred in the send processing after the remote program received send control.

3. 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 48 provides an example program using Set_Error_Direction.

"Send_Error (CMSERR)" on page 112 provides more information on reporting errors.

"Send-Pending State and the error_direction Characteristic" on page 182 provides more information on the *error_direction* characteristic.

Set_Fill (CMSF)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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** (input)*
Specifies the conversation identifier.

***fill** (input)*
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** (output)*
Specifies the result of the call execution. 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 call 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 97 provides more information on how the *fill* characteristic is used for basic conversations.

Set_Log_Data (CMSLD)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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.

For IMS and MVS, this call does update the *log_data* conversation characteristic. However, the error information is not logged or sent to the conversation partner in error situations.

Format

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

Parameters

conversation_ID* *(input)
Specifies the conversation identifier.

log_data* *(input)
Specifies the program-unique error information that is to be logged. The data supplied by the program is any data the program wants to have logged.

log_data_length* *(input)
Specifies the length of the program-unique error information. The length can be from 0 to 512 bytes. If zero, the *log_data_length* characteristic is set to zero (effectively setting the *log_data* characteristic to the null string), and the *log_data* parameter on this call is ignored.

return_code* *(output)
Specifies the result of the call execution. 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. If the *log_data* characteristic contains data (as a result of a Set_Log_Data call), log data will be sent to the remote LU under any of the following conditions:
   - When the local program issues a Send_Error call
   - When the local program issues a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND
   - When the local program issues a Send_Data call with *send_type* set to CM_SEND_AND_DEALLOCATE and *deallocate_type* set to CM_DEALLOCATE_ABEND.
2. The LU resets the *log_data* and *log_data_length* characteristics to their initial (null) values after sending the log data. Therefore, the *log_data* is sent to the remote LU only once even though an error indication may be issued several times. See usage note 1 for conditions when log data is sent.
3. Specify *log_data* using the local system's native encoding. When the log data is displayed on the partner system, it will be displayed in that system's native encoding.
4. 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

"Send_Error (CMSERR)" on page 112 and "Deallocate (CMDEAL)" on page 75 provide further discussion on how the *log_data* characteristic is used.

"Automatic Conversion of Characteristics" on page 20 provides further information on the automatic conversion of the *log_data* parameter.

Set_Mode_Name (CMSMN)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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** (input)*
Specifies the conversation identifier.

***mode_name** (input)*
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: A program must have special authority to specify a mode name that is used by SNA service transaction programs only, such as SNASVCMG.

***mode_name_length** (input)*
Specifies the length of the mode name. The length can be from zero to eight bytes. If zero, the mode name for this conversation is set to null and the *mode_name* parameter included with this call is not significant.

***return_code** (output)*
Specifies the result of the call execution. 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. Specify *mode_name* using the local system's native encoding. CPI Communications automatically converts the *mode_name* from the native encoding where necessary.
3. 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 183 provides a discussion of SNA service transaction programs.

"Side Information" on page 15 provides further discussion of the *mode_name* conversation characteristic.

"Automatic Conversion of Characteristics" on page 20 provides further information on the automatic conversion of the *mode_name* parameter.

Set_Partner_LU_Name (CMSPLN)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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 call) can issue Set_Partner_LU_Name.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

partner_LU_name *(input)*
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 *(input)*
Specifies the length of the partner LU name. The length can be from 1 to 17 bytes.

return_code *(output)*
Specifies the result of the call execution. 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 call does not cause a state change.

Usage Notes

1. Specify *partner_LU_name* using the local system's native encoding. CPI Communications automatically converts the *partner_LU_name* from the native encoding where necessary.
2. 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 15 and note 2 of Table 7 on page 153 provide further discussion of the *partner_LU_name* conversation characteristic.

"Automatic Conversion of Characteristics" on page 20 provides further information on the automatic conversion of the *partner_LU_name* parameter.

Set_Prepare_To_Receive_Type (CMSPTR)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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 call.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

prepare_to_receive_type *(input)*
Specifies the type of prepare-to-receive processing 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.
  - If *sync_level* is set to CM_SYNC_POINT, enter **Defer-Receive** state until the program issues an SAA resource recovery interface Commit or Backout call, or until the program issues a Confirm or Flush call for this conversation. If the Commit or Confirm call is successful or if a Flush call is issued, the conversation then enters **Receive** state. If the Backout call is successful, the conversation returns to its state at the previous sync point. Otherwise, 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.

return_code *(output)*

Specifies the result of the call execution. 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 93 provides a discussion of how the *prepare_to_receive_type* is used.

"Example 5: The Receiving Program Changes the Data Flow Direction" on page 44 shows an example program using the Prepare_To_Receive call.

Set_Receive_Type (CMSRT)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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 *(input)*
Specifies the conversation identifier.

receive_type *(input)*
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 *(output)*
Specifies the result of the call execution. 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 97 provides a discussion of how the *receive_type* characteristic is used.

"Example 3: The Sending Program Changes the Data Flow Direction" on page 40 provides a discussion of how a program can use Set_Receive_Type with a value of CM_RECEIVE_IMMEDIATE.

Set_Return_Control (CMSRC)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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 call after an Allocate has been issued for a conversation. Only the program that initiates the conversation (with the Initialize_Conversation call) can issue this call.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

return_control *(input)*
Specifies when a program receives 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 *(output)*
Specifies the result of the call execution. 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 a 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 67 provides more discussion on the use of the *return_control* characteristic in allocating a conversation.

Set_Send_Type (CMSST)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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 call.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

send_type *(input)*
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 might not be sent immediately but, instead, might 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 *(output)*

Specifies the result of the call execution. 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 107 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 88
- "Confirm (CMCFM)" on page 70
- "Prepare_To_Receive (CMPTR)" on page 93
- "Deallocate (CMDEAL)" on page 75

"Example 5: The Receiving Program Changes the Data Flow Direction" on page 44 shows an example program flow using the Set_Send_Type call.

Set_Sync_Level (CMSSL)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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 the programs support no synchronization, confirmation-level synchronization (using the Confirm and Confirmed CPI Communications calls), or sync-point-level synchronization (using SAA resource recovery interface 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 *(input)*
Specifies the conversation identifier.

sync_level *(input)*
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 or sync point processing on this conversation. The programs will not issue any calls or recognize any returned parameters relating to synchronization.
- CM_CONFIRM
  The programs can perform confirmation processing on this conversation. The programs can issue calls and recognize returned parameters relating to confirmation.
- CM_SYNC_POINT
  The programs can perform sync point processing on this conversation. The programs can issue SAA resource recovery interface calls and will recognize returned parameters relating to resource recovery interface processing. The programs can also perform confirmation processing.

The CM_SYNC_POINT value is not valid on CICS, IMS, MVS, OS/2, or OS/400 systems.

return_code *(output)*
Specifies the result of the call execution. 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.
  - The *sync_level* is set to CM_SYNC_POINT and the local system does not support the SAA resource recovery interface.
  - The *sync_level* is set to CM_SYNC_POINT and *deallocate_type* is set to CM_FLUSH or CM_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 70 and "Confirmed (CMCFMD)" on page 73 provide further information on confirmation processing.

"Example 4: Validation and Confirmation of Data Reception" on page 42 and "Example 8: Sending Program Issues a Commit" on page 50 show how to use the Set_Sync_Level call and how to perform confirm and sync point processing.

Set_TP_Name (CMSTPN)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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 15 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 *(input)*
Specifies the conversation identifier.

TP_name *(input)*
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 *(input)*
Specifies the length of *TP_name.* The length can be from 1 to 64 bytes.

return_code *(output)*
Specifies the result of the call execution. 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

1. Specify *TP_name* using the local system's native encoding. CPI Communications automatically converts the *TP_name* from the native encoding to EBCDIC encoding where necessary.
2. If a *return_code* other than CM_OK is returned on the call, the *TP_name* and *TP_name_length* conversation characteristics are unchanged.
3. The *TP_name* specified on this call must be formatted according to the naming conventions of the partner system.

Related Information

See "SNA Service Transaction Programs" on page 183 for more information on privilege and service transaction programs.

"Side Information" on page 15 and notes 2 and 5 of Table 7 on page 153 provide further discussion of the *TP_name* conversation characteristic.

"Automatic Conversion of Characteristics" on page 20 provides further information on the automatic conversion of the *TP_name* parameter.

Test_Request_To_Send_Received (CMTRTS)

MVS	VM	OS/400	OS/2	IMS	CICS
X	X	X	X	X	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 *(input)*
Specifies the conversation identifier.

request_to_send_received *(output)*
Specifies the variable containing 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 place its end of the conversation in **Receive** state, which will place the remote program's end of the conversation in **Send** state.
- CM_REQ_TO_SEND_NOT_RECEIVED
  A request-to-send notification has not been received.

Note: Unless *return_code* is set to CM_OK, the value of *request_to_send_received* is not meaningful.

return_code *(output)*
Specifies the result of the call execution. 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**, **Send-Pending**, **Defer-Receive**, or **Defer-Deallocate** state.
  - For a conversation with *sync_level* set to CM_SYNC_POINT, the program may be in the **Backout-Required** condition. The Test_Request_To_Send_Received call is not allowed while the program is in this condition.
- 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 remote program may issue the Request_To_Send call more times than are indicated to the local 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 105 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 and characteristic.
- The character sets used by CPI Communications.
- 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 22, 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 5 on page 148 provides a mapping from valid pseudonyms to integer values for each variable and characteristic.

Pseudonyms can also be used for integer values in program code by making use of equate or define statements. CPI Communications provides sample pseudonym files for each of the SAA languages. See the appropriate operating environment appendix for the names of the pseudonym files, and see Appendix L, "Pseudonym Files" on page 335 for sample pseudonym files for all of the SAA CPI languages. See Appendix K, "Sample Programs" on page 323 for an example of how a pseudonym file 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 5 (Page 1 of 2). Variables/Characteristics and Their Possible Values

Variable or Characteristic Name	Pseudonym Values	Integer Values
conversation_state	CM_INITIALIZE_STATE	2
	CM_SEND_STATE	3
	CM_RECEIVE_STATE	4
	CM_SEND_PENDING_STATE	5
	CM_CONFIRM_STATE	6
	CM_CONFIRM_SEND_STATE	7
	CM_CONFIRM_DEALLOCATE_STATE	8
	CM_DEFER_RECEIVE_STATE	9
	CM_DEFER_DEALLOCATE_STATE	10
	CM_SYNC_POINT_STATE	11
	CM_SYNC_POINT_SEND_STATE	12
	CM_SYNC_POINT_DEALLOCATE_STATE	13
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 5 (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_PIP_NOT_SPECIFIED_CORRECTLY	5
	CM_SECURITY_NOT_VALID	6
	CM_SYNC_LVL_NOT_SUPPORTED_LU	7
	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
	CM_DEALLOCATED_ABEND_SVC	30
	CM_DEALLOCATED_ABEND_TIMER	31
	CM_SVC_ERROR_NO_TRUNC	32
	CM_SVC_ERROR_PURGING	33
	CM_SVC_ERROR_TRUNC	34
	CM_TAKE_BACKOUT	100
	CM_DEALLOCATED_ABEND_BO	130
	CM_DEALLOCATED_ABEND_SVC_BO	131
	CM_DEALLOCATED_ABEND_TIMER_BO	132
	CM_RESOURCE_FAIL_NO_RETRY_BO	133
	CM_RESOURCE_FAILURE_RETRY_BO	134
	CM_DEALLOCATED_NORMAL_BO	135
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
	CM_TAKE_COMMIT	5
	CM_TAKE_COMMIT_SEND	6
	CM_TAKE_COMMIT_DEALLOCATE	7
sync_level	CM_NONE	0
	CM_CONFIRM	1
	CM_SYNC_POINT	2

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 and 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 graphic representations, are provided in Table 6. See *SNA Formats* for more information on character sets.

Table 6 (Page 1 of 2). Character Sets 01134 and 00640

Graphic	Description	Character Set	
		01134	00640
	Blank		X
.	Period		X
<	Less than sign		X
(	Left parenthesis		X
+	Plus sign		X
&	Ampersand		X
*	Asterisk		X
)	Right parenthesis		X
;	Semicolon		X
-	Dash		X
/	Slash		X
,	Comma		X
%	Percent sign		X
_	Underscore		X
>	Greater than sign		X
?	Question mark		X
:	Colon		X
'	Single quote		X
=	Equal sign		X
"	Double quote		X
a	Lowercase a		X
b	Lowercase b		X
c	Lowercase c		X
d	Lowercase d		X
e	Lowercase e		X
f	Lowercase f		X
g	Lowercase g		X
h	Lowercase h		X
i	Lowercase i		X
j	Lowercase j		X
k	Lowercase k		X
l	Lowercase l		X
m	Lowercase m		X
n	Lowercase n		X
o	Lowercase o		X
p	Lowercase p		X
q	Lowercase q		X
r	Lowercase r		X
s	Lowercase s		X

Table 6 (Page 2 of 2). Character Sets 01134 and 00640

Graphic	Description	Character Set	
		01134	00640
t	Lowercase t		X
u	Lowercase u		X
v	Lowercase v		X
w	Lowercase w		X
x	Lowercase x		X
y	Lowercase y		X
z	Lowercase z		X
A	Uppercase A	X	X
B	Uppercase B	X	X
C	Uppercase C	X	X
D	Uppercase D	X	X
E	Uppercase E	X	X
F	Uppercase F	X	X
G	Uppercase G	X	X
H	Uppercase H	X	X
I	Uppercase I	X	X
J	Uppercase J	X	X
K	Uppercase K	X	X
L	Uppercase L	X	X
M	Uppercase M	X	X
N	Uppercase N	X	X
O	Uppercase O	X	X
P	Uppercase P	X	X
Q	Uppercase Q	X	X
R	Uppercase R	X	X
S	Uppercase S	X	X
T	Uppercase T	X	X
U	Uppercase U	X	X
V	Uppercase V	X	X
W	Uppercase W	X	X
X	Uppercase X	X	X
Y	Uppercase Y	X	X
Z	Uppercase Z	X	X
0	Zero	X	X
1	One	X	X
2	Two	X	X
3	Three	X	X
4	Four	X	X
5	Five	X	X
6	Six	X	X
7	Seven	X	X
8	Eight	X	X
9	Nine	X	X

Variable Types

CPI Communications makes use of two variable types, integer and character string. Table 7 on page 153 defines the type and length of variables used in this document. Variable types are described 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 150, 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. CPI Communications defines two lengths for some character-string variables:

- **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 7 on page 153).
- **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 characters (also referred to as blank 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, such as discussed in note 4 on page 154 for the *partner_LU_name*, the concatenated character string as a whole is left-justified within the variable and the variable is filled out to the right with space characters. Space characters, if present, are not part of the concatenated character string.

Table 7. Variable Types and Lengths

Variable	**Variable Type**	**Character Set**	**Length**
buffer [1]	Character string	no restriction	0-32767 bytes
conversation_ID	Character string	no restriction	8 bytes
conversation_state	Integer	N/A	32 bits
conversation_type	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 [2]	Character string	00640	0-512 bytes
log_data_length	Integer	N/A	32 bits
mode_name [2, 3, 7]	Character string	01134	0-8 bytes
mode_name_length	Integer	N/A	32 bits
partner_LU_name [2, 3, 4]	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
request_to_send_received	Integer	N/A	32 bits
requested_length	Integer	N/A	32 bits
return_code	Integer	N/A	32 bits
return_control	Integer	N/A	32 bits
send_length	Integer	N/A	32 bits
send_type	Integer	N/A	32 bits
status_received	Integer	N/A	32 bits
sym_dest_name [2, 6]	Character string	01134	8 bytes
sync_level	Integer	N/A	32 bits
TP_name [2, 5]	Character string	00640	1-64 bytes
TP_name_length	Integer	N/A	32 bits

Notes:

1. When a transaction program is in conversation with another transaction program executing in an unlike environment (for example, an EBCDIC-environment program in conversation with an ASCII-environment program), *buffer* may require conversion from one encoding to the other. This conversion is the responsibility of the transaction program and is not currently provided by CPI Communications.

2. Specify these fields using the native encoding of the local system. When appropriate, CPI Communications automatically converts these fields to their EBCDIC representations when they are used as input parameters in a non-EBCDIC environment. When CPI Communications returns these fields to the program (for instance, as output parameters on one of the Extract calls), they are returned in the native encoding of the local system. See "Automatic Conversion of Characteristics" on page 20 for more information on automatic conversion of these fields.

3. 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.

   Note: Although IBM LU 6.2 product implementations use character set 00640 for the *mode_name* and *partner_LU_name* fields, it is recommended that character set 01134, which is a subset of character set 00640, be used instead to enhance program portability.

   The IMS and MVS implementations of CPI Communications allow use of the $, @, and # national characters in the *mode_name* and *partner_LU_name* fields and restrict the first character in these fields to an alphabetic or national character.

   The OS/2 implementation allows specification of either an alias or network name for the *partner_LU_name* variable. OS/2 distinguishes the specification of an alias name from a network name based on the absence or presence of the period character in the name. See Appendix H, "CPI Communications on OS/2" on page 201 for more information about alias and network names used with OS/2.

4. 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.

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

   On VM, a space is used as a delimiter instead of a period.

5. The following usage notes apply when specifying the *TP_name*:

   - The space character is not allowed in *TP_name*.
   - 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 181 and "SNA Service Transaction Programs" on page 183 for details.

- On IMS and MVS systems, IBM recommends that the asterisk (*) be avoided in TP names because it causes a list request when it is entered on panels of the APPC/MVS administration dialog. The comma should also be avoided in IMS and MVS TP names, because it acts as a parameter delimiter in DISPLAY APPC commands.

- The OS/2 implementation allows the use of characters outside character set 00640 for the *TP_name* variable. These include all characters allowed as an OS/2 file name. See Appendix H, "CPI Communications on OS/2" on page 201 for more information about default TP names used with OS/2.

6. The field containing the *sym_dest_name* parameter on the CMINIT call must be at least eight bytes long. The symbolic destination name within that field may be from 0 to 8 characters long, with its characters taken from character set 01134. If the symbolic destination name is shorter than eight characters, it should be left-justified in the variable field, and padded on the right with spaces. A *sym_dest_name* parameter composed of eight spaces has special significance. See "Initialize_Conversation (CMINIT)" on page 90 for more information.

7. The four names in the following list are IBM-defined mode names for user sessions and may be specified for CPI Communications conversations on systems where they are defined, even though they contain the character #, which is not found in character set 01134:

   - #BATCH
   - #BATCHSC
   - #INTER
   - #INTERSC

Appendix B. Return Codes

All calls have a parameter called *return_code* that is passed back to the program at the completion of a call. The return code can be used to determine call-execution results and any state change that may have occured on the specified conversation. On some calls, the return code is not the only source of call-execution information. For example, on the Receive call, the *status_received* and *data_received* parameters should also be checked.

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.

For programs using conversations with *sync_level* set to CM_SYNC_POINT, all return codes indicating a required backout have numeric values equal to or greater than CM_TAKE_BACKOUT. This allows the CPI Communications programmer to test for a range of return code values to determine if backout processing is required. An example is:

return_code > = CM_TAKE_BACKOUT

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, "Call 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 5 on page 148 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 session limit for the specified LU-name and mode-name pair is 0, or because of a system definition error or a session-activation protocol error. This return code is also returned when the session is deactivated because of a session protocol error before the conversation can be allocated. The program should not retry the allocation request until the condition is corrected. This return code is returned on the 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

This return code may be returned under one of the following conditions:

- The remote program issued a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND, 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.
- The remote transaction program terminated normally but did not deallocate the conversation before terminating. Node services at the remote LU deallocated the conversation on behalf of the remote transaction program.

This return code is reported to the local program on a call the program issues for a conversation in **Send** or **Receive** state. The conversation is now in **Reset** state.

CM_DEALLOCATED_ABEND_BO

This return code is returned only for conversations with *sync_level* set to CM_SYNC_POINT.

The remote program issued a Deallocate call with *deallocate_type* set to CM_DEALLOCATED_ABEND, 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. The conversation is now in **Reset** state.

The local transaction program is in the **Backout_Required** condition and must issue an SAA resource recovery interface Backout call in order to restore all protected resources to their status as of the last synchronization point.

CM_DEALLOCATED_ABEND_SVC

This return code may be returned under one of the following conditions:

- The remote transaction program, using an LU 6.2 application programming interface and not using CPI Communications, issued a DEALLOCATE verb specifying a TYPE parameter of ABEND_SVC. If the conversation for the remote program was in **Receive** state when the verb was issued, information sent by the local program and not yet received by the remote program is purged.
- The remote transaction program either terminated abnormally or terminated normally but did not deallocate the conversation before terminating. Node services at the remote LU deallocated the conversation on behalf of the remote transaction program.

This return code is reported to the local program on a call the program issues for a conversation in **Send** or **Receive** state. The conversation is now in **Reset** state.

CM_DEALLOCATED_ABEND_SVC_BO

This return code is returned only for conversations with *sync_level* set to CM_SYNC_POINT. It is returned under the same conditions described under CM_DEALLOCATED_ABEND_SVC above.

This return code is reported to the local program on a call the program issues for a conversation in **Send** or **Receive** state. The conversation is now in **Reset** state.

The local transaction program is in the **Backout_Required** condition and must issue an SAA resource recovery interface Backout call in order to restore all protected resources to their status as of the last synchronization point.

CM_DEALLOCATED_ABEND_TIMER

This return code is returned only when the remote transaction program is using an LU 6.2 application programming interface and is not using CPI Communications.

The remote LU 6.2 transaction program issued a DEALLOCATE verb specifying a TYPE parameter of ABEND_TIMER. If the conversation for the remote program was in **Receive** state when the verb was issued, information sent by the local program and not yet received by the remote program is purged. This return code is reported to the local program on a call the program issues for a conversation in **Send** or **Receive** state. The conversation is now in **Reset** state.

CM_DEALLOCATED_ABEND_TIMER_BO

This return code is returned only for conversations with *sync_level* set to CM_SYNC_POINT, and only when the remote transaction program is using an LU 6.2 application programming interface and is not using CPI Communications.

The remote LU 6.2 transaction program issued a DEALLOCATE verb specifying a TYPE parameter of ABEND_TIMER. If the conversation for the remote program was in **Receive** state when the verb was issued, information sent by the local program and not yet received by the remote program is purged. This return code is reported to the local program on a call the program issues for a conversation in **Send** or **Receive** state. The conversation is now in **Reset** state.

The local transaction program is in the **Backout_Required** condition and must issue an SAA resource recovery interface Backout call in order to restore all protected resources to their status as of the last synchronization point.

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 *sync_level* is CM_NONE. This return code is reported to the local program on a call the program issues for a conversation in **Receive** state. The conversation is now in **Reset** state.

CM_DEALLOCATED_NORMAL_BO

This return code is returned only for conversations with *sync_level* set to CM_SYNC_POINT.

When the Send_Error call is issued in **Receive** state, incoming information is purged by the LU. This purged information may include an abend deallocation notification from the remote program or LU. When such a notification is purged, CPI Communications returns CM_DEALLOCATED_NORMAL_BO instead of one of the following return codes:

- CM_DEALLOCATED_ABEND_BO
- CM_DEALLOCATED_ABEND_SVC_BO
- CM_DEALLOCATED_ABEND_TIMER_BO

The conversation is now in **Reset** state.

The local transaction program is in the **Backout_Required** condition and must issue an SAA resource recovery interface Backout call in order to restore all protected resources to their status as of the last synchronization point.

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_PIP_NOT_SPECIFIED_CORRECTLY

This return code is returned only when the remote transaction program is using an LU 6.2 application programming interface and is not using CPI Communications.

The remote LU 6.2 rejected the allocation request because the remote program has one or more program initialization parameter (PIP) variables defined. CPI Communications does not support these parameters. This return code is returned on a call made after the Allocate. When this return code is returned to the program, the conversation is in **Reset** state.

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 for a conversation 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 for a conversation 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). This return code may be caused by the failure of the program to pass a valid parameter address. 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

This return code may be returned under one of the following conditions:

- The local program issued a call for a conversation in a state that was not valid for that call.
- The *sync_level* is set to CM_SYNC_POINT, the local program is in the **Backout-Required** condition, and the call issued is not valid while the program is in this condition.

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

This return code may be returned under one of the following conditions:

- 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.
- The remote transaction program terminated normally but did not deallocate the conversation before terminating. Node services at the remote LU deallocated the conversation on behalf of the remote transaction program.

This return code can be reported to the local program on a call it issues for a conversation in any state other than **Reset** or **Initialize**. The conversation is in **Reset** state.

CM_RESOURCE_FAIL_NO_RETRY_BO

This return code is returned only for conversations with *sync_level* set to CM_SYNC_POINT.

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 for a conversation in any state other than **Reset** or **Initialize**. The conversation is in **Reset** state.

The local transaction program is in the **Backout_Required** condition and must issue an SAA resource recovery interface Backout call in order to restore all protected resources to their status as of the last synchronization point.

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 for a conversation in any state other than **Reset** or **Initialize**. The conversation is in **Reset** state.

CM_RESOURCE_FAILURE_RETRY_BO
This return code is returned only for conversations with *sync_level* set to CM_SYNC_POINT.

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 for a conversation in any state other than **Reset** or **Initialize**. The conversation is in **Reset** state.

The local transaction program is in the **Backout_Required** condition and must issue an SAA resource recovery interface Backout call in order to restore all protected resources to their status as of the last synchronization point.

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_SVC_ERROR_NO_TRUNC
This return code is returned only when the remote transaction program is using an LU 6.2 application programming interface and is not using CPI Communications.

The remote LU 6.2 transaction program issued a Send_Error verb specifying a TYPE parameter of SVC, the conversation for the remote program was in **Send** state, and the verb did not truncate a logical record. This return code is returned on a Receive call. When this return code is returned to the local program, the conversation is in **Receive** state.

CM_SVC_ERROR_PURGING
This return code is returned only when the remote transaction program is using an LU 6.2 application programming interface and is not using CPI Communications.

The remote LU 6.2 transaction program issued a Send_Error verb specifying a TYPE parameter of SVC; the conversation for the remote program was in **Receive**, **Confirm**, or **Sync-Point** state; and the verb may have caused information to be purged. This return code is normally reported to the local program on a call the local program issues after sending some information to the remote program. However, the return code can be reported on a call the local program issues before sending any information, depending on the call and when it is issued. When this return code is returned to the local program, the conversation is in **Receive** state.

CM_SVC_ERROR_TRUNC
This return code is returned only when the remote transaction program is using an LU 6.2 application programming interface and is not using CPI Communications.

The remote LU 6.2 transaction program issued a Send_Error verb specifying a TYPE parameter of SVC, the conversation for the remote program was in **Send** state, and the verb truncated a logical record. Truncation occurs at the basic conversation protocol boundary when a program begins sending a logical record and then issues Send_Error before sending the complete logical record. This return code is reported to the local program on a Receive call the local program issues after receiving the truncated logical record. The conversation remains in **Receive** state.

CM_SYNC_LVL_NOT_SUPPORTED_LU
This return code is returned only for conversations with *sync_level* set to CM_SYNC_POINT.

The local system rejected the allocation request because the local program specified a *sync_level* of CM_SYNC_POINT, which the remote system does not support. This return code is returned on the Allocate call. 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_TAKE_BACKOUT
This return code is returned only for conversations with *sync_level* set to CM_SYNC_POINT.

The remote program, the local system, or the remote system issued an SAA resource recovery interface Backout call, and the local application must issue a Backout call in order to restore all protected resources to their status as of the last synchronization point. The local program is in the **Backout-Required** condition upon receipt of this return code. Once the local program issues the Backout call, the conversation is placed in the state it was in at the time of the last sync point operation.

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 21, 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 8 on page 175 describes the state transitions that are allowed for the CPI Communications calls.

Table 9 on page 179 shows the effects of SAA resource recovery interface Commit and Backout calls on CPI Communications conversation states.

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 parentheses — (. . .)
- **return_code** abbreviations are enclosed by brackets — [. . .]
- **data_received and status_received** abbreviations are enclosed by braces and separated by a comma — { . . . , . . . }. 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_LEVEL and *sync_level* is set to CM_CONFIRM For a Prepare_To_Receive call, 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(x)	*send_type* is set to CM_SEND_AND_DEALLOCATE. *x* represents the *deallocate_type* and can be A, C, F, or S. Refer to the appropriate entries in this table for a description of these values.
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(x)	*send_type* is set to CM_SEND_AND_PREP_TO_RECEIVE. *x* represents the *prepare_to_receive_type* and can be C, F, or S. Refer to the appropriate entries in this table for a description of these values.
S	For a Deallocate call, S means the following: • *deallocate_type* is set to CM_DEALLOCATE_SYNC_LVL and *sync_level* is set to CM_SYNC_POINT For a Prepare_To_Receive call, S means the following: • *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* is set to CM_SYNC_POINT
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 • CM_SYNC_LVL_NOT_SUPPORTED_LU For any other call, ae means one of the following: • CM_CONVERSATION_TYPE_MISMATCH • CM_PIP_NOT_SPECIFIED_CORRECTLY • 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
bo	CM_TAKE_BACKOUT. This return code is returned only for conversations with *sync_level* set to CM_SYNC_POINT.
da	da means one of the following: • CM_DEALLOCATED_ABEND • CM_DEALLOCATED_ABEND_SVC • CM_DEALLOCATED_ABEND_TIMER
db	db is returned only for conversations with *sync_level* set to CM_SYNC_POINT and means one of the following: • CM_DEALLOCATED_ABEND_BO • CM_DEALLOCATED_ABEND_SVC_BO • CM_DEALLOCATED_ABEND_TIMER_BO
dn	CM_DEALLOCATED_NORMAL
dnb	CM_DEALLOCATED_NORMAL_BO. This return code is returned only for conversations with *sync_level* set to CM_SYNC_POINT.
en	en means one of the following: • CM_PROGRAM_ERROR_NO_TRUNC • CM_SVC_ERROR_NO_TRUNC
ep	ep means one of the following: • CM_PROGRAM_ERROR_PURGING • CM_SVC_ERROR_PURGING
et	et means one of the following: • CM_PROGRAM_ERROR_TRUNC • CM_SVC_ERROR_TRUNC
ok	CM_OK
pc	CM_PROGRAM_PARAMETER_CHECK. This return code means an error was found in one or more parameters. For calls illegally issued in **Reset** state, pc is returned because the *conversation_ID* is undefined in that state.
pe	CM_PARAMETER_ERROR
rb	rb means one of the following: • CM_RESOURCE_FAIL_NO_RETRY_BO • CM_RESOURCE_FAILURE_RETRY_BO

Abbreviation	Meaning
rf	rf means one of the following: • CM_RESOURCE_FAILURE_NO_RETRY • CM_RESOURCE_FAILURE_RETRY
sc	CM_PROGRAM_STATE_CHECK
un	CM_UNSUCCESSFUL

Note: The return code CM_PRODUCT_SPECIFIC_ERROR is not included in the state table because the state transitions caused by this return code are product-specific. See the appropriate product appendix for a description of these state transitions.

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
co	CM_CONFIRM_RECEIVED
cs	CM_CONFIRM_SEND_RECEIVED
no	CM_NO_STATUS_RECEIVED
se	CM_SEND_RECEIVED
tc	CM_TAKE_COMMIT. This value is returned only for conversations with *sync_level* set to CM_SYNC_POINT.
td	CM_TAKE_COMMIT_DEALLOCATE. This value is returned only for conversations with *sync_level* set to CM_SYNC_POINT.
ts	CM_TAKE_COMMIT_SEND. This value is returned only for conversations with *sync_level* set to CM_SYNC_POINT.

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. CPI Communications either will not allow this input or will never return the indicated return codes for this input in this state.
–	Remain in current state
1-13	Number of next state
↓	It is valid to make this call from this state. See the table entries immediately below this symbol to determine the state transition resulting from the call.
↓′	For programs not using *sync_level* set to CM_SYNC_POINT, this is equivalent to ↓. If the conversation has *sync_level* set to CM_SYNC_POINT, however, ↓′ means it is valid to make this call from this state *unless the transaction program is in the* **Backout-Required** *condition*. In that case, the call is invalid and CM_PROGRAM_STATE_CHECK is returned. For valid calls, see the table entries immediately below this symbol to determine the state transition resulting from the call.
^	For programs not using *sync_level* set to CM_SYNC_POINT, this symbol should be ignored. For programs using *sync_level* set to CM_SYNC_POINT, when this symbol follows a state number or a – (for example, 1^ or –^), it means the program may be in the **Backout-Required** condition following the call.
#	Conversations with *sync_level* set to CM_SYNC_POINT go to the state they were in at the completion of the most recent synchronization point. If there was no prior synchronization event, the side of the conversation that was initialized with an Allocate call goes to **Send** state, and the side of the conversation that had an Accept_Conversation call goes to **Receive** state.

How to Use the State Table

Each CPI Communications call is represented in the table by a group of input rows. The possible conversation states are shown across the top of the table. The states correspond to the columns of the matrix. The intersection of input (row) and state (column) represents the validity of a CPI Communications call in that particular state and, for valid calls, what state transition (if any) occurs.

The first row of each call input grouping (delineated by horizontal lines) contains the name of the call and a symbol in each state column showing whether the call is valid for that state. A call is valid for a given state only if that state's column contains a downward pointing arrow (↓) on this row. If the [sc] or [pc] symbol appears in a state's column, the call is invalid for that state and receives a return code of CM_PROGRAM_STATE_CHECK or CM_PROGRAM_PARAMETER_CHECK, respectively. No state transitions occur for invalid CPI Communications calls.

The remaining input rows in the call group show the state transitions for valid calls. The transition from one conversation state to another often depends on the value of the return code returned by the call; therefore, a given call group may have several rows, each showing the state transitions for a particular return code or set of return codes.

For conversations with *sync_level* set to CM_SYNC_POINT, the following special considerations apply:

- A state transition symbol ending with a carat (e.g., 1^ or —^) means that the conversation's transaction program may be in the **Backout-Required** condition following the call. (Note that the state change for the conversation is indicated by the **first** character of these symbols.)
- When a transaction program is in the **Backout-Required** condition, its protected conversations are restricted from issuing certain CPI Communications calls. These calls are designated in the table with the symbol ↓′. Where this symbol appears, the call is valid in this state unless the conversation is protected and the program is in the **Backout-Required** condition. If the call is invalid, a *return_code* of CM_PROGRAM_STATE_CHECK is returned and no conversation state transition occurs.
- Table 9 on page 179 shows the effects of SAA resource recovery interface Commit and Backout calls on CPI Communications conversation states.

Example

For an example of how the state table might be used, look at the group of input rows for the **Deallocate(C)** call. The **(C)** here means that this group is for the Deallocate call when either *deallocate_type* is set to CM_DEALLOCATE_CONFIRM or *deallocate_type* is set to CM_DEALLOCATE_SYNC_LVL and *sync_level* is set to CM_CONFIRM. The first row in this group shows that this call is valid only when the conversation is in **Send** or **Send-Pending** state. For all other states, either the call is invalid and a *return_code* of CM_PROGRAM_PARAMETER_CHECK or CM_PROGRAM_STATE_CHECK is returned, or the call is not possible.

Beneath the input row containing **Deallocate(C)**, there are three rows showing the possible return codes returned by this call. Since the call is valid only in **Send** and **Send-Pending** states, only these states' columns contain transition values on these three rows. These transition values provide the following information:

- The conversation goes from **Send** or **Send-Pending** state to **Reset** state (state 1) when a return code abbreviated as "ok," "da," or "rf" is returned. See "Return Code Values [ ]" on page 169 to find out what these abbreviations mean.
- The conversation goes from **Send** state to **Reset** state when a return code abbreviated as "ae" is returned.
- A return code abbreviated as "ae" will never be returned when this call is issued from **Send-Pending** state.
- The conversation goes from **Send** or **Send-Pending** state to **Receive** state (state 4) when a return code abbreviated as "ep" is returned.
- There is no state transition when a return code of CM_PROGRAM_PARAMETER_CHECK ("pc") is returned.

Table 8 (Page 1 of 4). States and Transitions for CPI Communications Calls

Inputs	Used by all conversations								Used only by conversations with *sync_level* set to CM_SYNC_POINT 1				
	Reset	Initialize	Send	Receive	Send-Pending	Confirm	Confirm-Send	Confirm-Deallocate	Defer-Receive	Defer-Deallocate	Sync-Point	Sync-Point Send	Sync-Point Deallocate
	1	2	3	4	5	6	7	8	9	10	11	12	13
Accept_Conversation	↓	/	/	/	/	/	/	/	/	/	/	/	/
[ok]	4												
[sc]	–												
Allocate	[pc]	↓'	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]		3											
[ae]		1											
[pc,pe,un]		–											
Confirm	[pc]	[sc]	↓'	[sc]	↓'	[sc]	[sc]	[sc]	↓'	[sc]	[sc]	[sc]	[sc]
[ok]			–		3				4				
[ae]			1		/				1				
[da,rf]			1		1				1				
[bo]			–^		3^				4^				
[db,rb]			1^		1^				1^				
[ep]			4		4				4				
[pc]			–		–				–				
Confirmed	[pc]	[sc]	[sc]	[sc]	[sc]	↓'	↓'	↓'	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]						4	3	1					
[pc]						–	–	–					
Deallocate(A)	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok]		1^	1^	1^	1^	1^	1^	1^	1^	1^	1^	1^	1^
[pc]		–	–	–	–	–	–	–	–	–	–	–	–
Deallocate(C)	[pc]	[sc]	↓	[sc]	↓	[sc]	[sc]	[sc]	/	/	/	/	/
[ok,da,rf]			1		1								
[ae]			1		/								
[ep]			4		4								
[pc]			–		–								
Deallocate(F)	[pc]	[sc]	↓	[sc]	↓	[sc]	[sc]	[sc]	/	/	/	/	/
[ok]			1		1								
[pc]			–		–								
Deallocate(S)	[pc]	[sc]	↓'	[sc]	↓'	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]			10		10								
[pc]			–		–								
Extract_Conv_State	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–
[bo]		/	–	–	–	–	–	–	–	–	–	–	–
Extract_Conv_Type	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–
Extract_Mode_Name	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–
Extract_Part_LU_Name	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–
Extract_Sync_Level	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–
Flush	[pc]	[sc]	↓'	[sc]	↓'	[sc]	[sc]	[sc]	↓'	[sc]	[sc]	[sc]	[sc]
[ok]			–		3				4				
[pc]			–		–				–				

Table 8 (Page 2 of 4). States and Transitions for CPI Communications Calls

Inputs	Used by all conversations								Used only by conversations with *sync_level* set to CM_SYNC_POINT [1]				
	Reset	Initialize	Send	Receive	Send-Pending	Confirm	Confirm-Send	Confirm-Deallocate	Defer-Receive	Defer-Deallocate	Sync-Point	Sync-Point Send	Sync-Point Deallocate
	1	2	3	4	5	6	7	8	9	10	11	12	13
Initialize_Conversation[2]	↓	/	/	/	/	/	/	/	/	/	/	/	/
[ok]	2												
[pc]	—												
Prepare_To_Receive(C)	[pc]	[sc]	↓'	[sc]	↓'	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,ep]			4		4								
[ae]			1		/								
[da,rf]			1		1								
[bo]			4^		4^								
[db,rb]			1^		1^								
[pc]			—		—								
Prepare_To_Receive(F)	[pc]	[sc]	↓'	[sc]	↓'	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]			4		4								
[pc]			—		—								
Prepare_To_Receive(S)	[pc]	[sc]	↓'	[sc]	↓'	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]			9		9								
[pc]			—		—								
Receive(I)	[pc]	[sc]	[sc]	↓'	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok] {dr,no}				—									
[ok] {nd,se}				3									
[ok] {dr,se}				5									
[ok] {*,co}				6									
[ok] {*,cs}				7									
[ok] {*,cd}				8									
[ok] {*,tc}				11									
[ok] {*,ts}				12									
[ok] {*,td}				13									
[ae,da,dn,rf]				1									
[bo]				—^									
[db,rb]				1^									
[en,ep,et,pc,un]				—									
Receive(W)	[pc]	[sc]	↓'	↓'	↓'	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok] {dr,no}			4	—	4								
[ok] {nd,se}			—	3	3								
[ok] {dr,se}			5	5	—								
[ok] {*,co}			6	6	6								
[ok] {*,cs}			7	7	7								
[ok] {*,cd}			8	8	8								
[ok] {*,tc}			11	11	11								
[ok] {*,ts}			12	12	12								
[ok] {*,td}			13	13	13								
[ae]			1	1	/								
[bo]			4^	—^	4^								
[da,dn,rf]			1	1	1								
[db,rb]			1^	1^	1^								
[en,ep]			4	—	4								
[et]			/	—	/								
[pc]			—	—	—								
Request_To_Send	[pc]	[sc]	↓'	↓'	↓'	↓'	↓'	↓'	[sc]	[sc]	↓'	↓'	↓'
[ok,pc]			—	—	—	—	—	—			—	—	—

Table 8 (Page 3 of 4). States and Transitions for CPI Communications Calls

Inputs	Used by all conversations								Used only by conversations with *sync_level* set to CM_SYNC_POINT 1				
	Reset	Initialize	Send	Receive	Send-Pending	Confirm	Confirm-Send	Confirm-Deallocate	Defer-Receive	Defer-Deallocate	Sync-Point	Sync-Point Send	Sync-Point Deallocate
	1	2	3	4	5	6	7	8	9	10	11	12	13
Send_Data	[pc]	[sc]	↓′	[sc]	↓′	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
(B) [ok]			–		3								
(C) [ok]			–		3								
(F) [ok]			–		3								
(P(C)) [ok]			4		4								
(P(F)) [ok]			4		4								
(P(S)) [ok]			9		9								
(D(A)) [ok]			1^		1^								
(D(C)) [ok]			1		1								
(D(F)) [ok]			1		1								
(D(S)) [ok]			10		10								
[ae]			1		/								
[da,rf]			1		1								
[bo]			–^		3^								
[db,rb]			1^		1^								
[ep]			4		4								
[pc]			–		–								
Send_Error	[pc]	[sc]	↓′	↓′	↓′	↓′	↓′	↓′	[sc]	[sc]	↓′	↓′	↓′
[ok]			–	3	3	3	3	3			3	3	3
[ae,da]			1	/	/	/	/	/			/	/	/
[bo]			–^	/	3^	/	/	/			/	/	/
[db]			1^	/	/	/	/	/			/	/	/
[dn]			/	1	/	/	/	/			/	/	/
[dnb]			/	1^	/	/	/	/			/	/	/
[ep]			4	/	/	/	/	/			/	/	/
[pc]			–	–	–	–	–	–			/	/	/
[rb]			1^	1^	1^	1^	1^	1^			1^	1^	1^
[rf]			1	1	1	1	1	1			1	1	1
Set_Conversation_Type	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–											
Set_Deallocate_Type	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–
Set_Error_Direction	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–
Set_Fill	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–
Set_Log_Data	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–
Set_Mode_Name	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–											
Set_Partner_LU_Name	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–											
Set_Prep_To_Rcv_Type	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–
Set_Receive_Type	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–
Set_Return_Control	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–											

Table 8 (Page 4 of 4). States and Transitions for CPI Communications Calls

Inputs	Used by all conversations								Used only by conversations with *sync_level* set to CM_SYNC_POINT [1]				
	Reset 1	Initialize 2	Send 3	Receive 4	Send-Pending 5	Confirm 6	Confirm-Send 7	Confirm-Deallocate 8	Defer-Receive 9	Defer-Deallocate 10	Sync-Point 11	Sync-Point Send 12	Sync-Point Deallocate 13
Set_Send_Type	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–
Set_Sync_Level	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–											
Set_TP_Name	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–											
Test_Req_to_Send_Rcvd	[pc]	[sc]	↓′	↓′	↓′	[sc]	[sc]	[sc]	↓′	↓′	[sc]	[sc]	[sc]
[ok,pc]			–	–	–				–	–			

Effects of Calls to the SAA Resource Recovery Interface

Table 9 on page 179 shows the state transitions resulting from calls to the resource recovery interface. This table applies only to conversations with *sync_level* set to CM_SYNC_POINT.

The following abbreviations are used for return codes in Table 9:

Abbreviation	Meaning
bo	RR_BACKED_OUT
bom	RR_BACKED_OUT_OUTCOME_MIXED
bop	RR_BACKED_OUT_OUTCOME_PENDING
com	RR_COMMITTED_OUTCOME_MIXED
cop	RR_COMMITTED_OUTCOME_PENDING
ok	RR_OK
sc	RR_STATE_CHECK

[1] Because CICS, IMS, MVS, OS/2, and OS/400 do not support the use of *sync_level*=CM_SYNC_POINT, states 9–13 and transitions to and from those states are not applicable on these systems.

[2] **While the Initialize_Conversation call can be issued only once for any given conversation, a program can issue multiple Initialize_Conversation calls to establish concurrent conversations with different partners. For more information, see "Multiple Conversations" on page 24.**

Table 9. States and Transitions for Protected Conversations

	Reset	Initialize	Send	Receive	Send-Pending	Confirm	Confirm-Send	Confirm-Deallocate	Defer-Receive	Defer-Deallocate	Sync-Point	Sync-Point Send	Sync-Point Deallocate
Inputs	1	2	3	4	5	6	7	8	9	10	11	12	13
Commit call	[sc][3]	↓[5]	↓	[sc]	↓	[sc]	[sc]	[sc]	↓	↓	↓	↓	↓
[ok,cop,com]		–	–		3				4	1	4	3	1
[bo,bop,bom]		–	#		#				#	#	#	#	#
[sc]		–	–		–				–	–	–	–	–
Backout call	↓[4]	↓[5]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,bop,bom]	–	–	#	#	#	#	#	#	#	#	#	#	#

[3] When a program started by an incoming allocation request issues a Commit call before issuing an Accept_Conversation call, a state check results. The Commit call has no effect on other conversations in Reset state.

[4] When a program started by an incoming allocation request issues a Backout call before issuing an Accept_Conversation call, the underlying LU 6.2 conversation is actually backed out, though the CPI Communications conversation remains in **Reset** state.

[5] Conversations in **Initialize** state are not affected by Commit and Backout calls.

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 10 beginning on page 184 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.

Note: Although much of the LU 6.2 function has been included in CPI Communications, some of the function has not. Likewise, CPI Communications contains features that are not found in LU 6.2. These features are differences in syntax. The semantics of LU 6.2 function have not been changed or extended.

CPI Communications contains the following features not found in LU 6.2:

- The Initialize_Conversation call and side information, used to initialize conversation characteristics without requiring the application program to explicitly specify these parameters.
- A conversation state of **Send-Pending** (discussed in more detail in "Send-Pending State and the error_direction Characteristic" on page 182).
- 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 182).
- 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.

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. However, such a method may be provided as a CPI Communications extension in some environments. See the appropriate product appendix for information about a particular environment.

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 used in CPI Communications to eliminate ambiguity about the source of some errors. A program using CPI Communications can 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.

The ambiguity is eliminated in CPI Communications by use of the **Send-Pending** state and *error_direction* characteristic. CPI Communications places the conversation 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 conversation is in **Send-Pending** state and the program issues a Send_Error call, 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 48.

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 naming conventions for 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 wants to specify an SNA service transaction program, the character set shown for *TP_name* in Appendix A, "Variables and Characteristics" is inadequate. The first character of an SNA service transaction program name is a character with a value in the range from X'00' through X'0D' or 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 service 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.

Note: Because of the special nature of SNA service transaction program names, they cannot be specified on the Set_TP_Name call in a non-EBCDIC environment. A CPI Communications program in a non-EBCDIC environment wanting to establish a conversation with an SNA service transaction program must ensure that the desired *TP_name* is included in the side information.

Relationship between LU 6.2 Verbs and CPI Communications Calls

Table 10 beginning on page 184 shows 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 or verb parameter row and a 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 10 (Page 1 of 3). Relationship of LU 6.2 Verbs to CPI Communications Calls

CPI Communications Calls

	Starter Set						Advanced Function																							
LU 6.2 Verbs	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				D																										
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 10 (Page 2 of 3). Relationship of LU 6.2 Verbs to CPI Communications Calls

	CPI Communications Calls																													
	Starter Set						**Advanced Function**																							
LU 6.2 Verbs	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																														
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				D																										
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 10 (Page 3 of 3). Relationship of LU 6.2 Verbs to CPI Communications Calls

CPI Communications Calls

	Starter Set						Advanced Function																							
LU 6.2 Verbs	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. CPI Communications on CICS/ESA

CICS/ESA Version 3 Release 2 provides support for CPI Communications. This appendix summarizes the product-specific information that the CICS programmer needs when working with CPI Communications. The information is presented under these headings:

- Usage Guidelines
- Language Support and Programming Interfaces
- Side Information
- Special CICS/ESA Notes

The *CICS/ESA Intercommunication Guide* deals with CICS intercommunication generally, and gives details of how to define links for APPC support.

Usage Guidelines

CPI Communications in CICS provides an alternative application programming interface (API) to existing CICS LU 6.2 communications support.

Users who have already made a skill investment in the existing EXEC CICS programming interface, or who do not expect to require the cross system consistency benefits offered by SAA, may choose to continue using the EXEC CICS API. Investment in applications written to the CICS API is protected by IBM's continuing commitment to its enhancement and support across all CICS products.

Alternatively, users may prefer to use CPI Communications in APPC networks that include multiple system platforms, where the consistency of a common API is seen to be of benefit.

A major benefit of the common LU 6.2 standard is that applications which use CPI Communications can converse with applications on any system that provides an LU 6.2 API. This includes applications on different CICS platforms. However, there are some restrictions which the user should be aware of regarding LU 6.2 partners of CPI Communications programs. These are documented in Appendix D of this manual.

A CICS transaction program can use both CICS LU 6.2 API commands and CPI Communications calls in the same program, but may **not** use both in the same conversation.

Language Support and Programming Interfaces

CICS/ESA supports the SAA languages COBOL, PL/I, and C for use with CPI Communications. 370 Assembler, although not an SAA language, is also supported.

The programming interfaces for COBOL, PL/I, and C are as described in "Programming Language Considerations" on page 59. All programs must set up a parameter list and obey the standard IBM linking convention.

CICS provides a pseudonym file for each language supported:

Table 11. CICS Pseudonym Files for Supported Languages

Language	File name	Location
COBOL	CMCOBOL	CICSxxx.COBLIB
PL/I	CMPLI	CICSxxx.PL1LIB
C	CMC	CICSxxx.CEELIB
Assembler	CMHASM	CICSxxx.MACLIB

Note: xxx is the CICS release number. (For example, 321 would mean Version 3 Release 2 Modification 1.)

Side Information

CICS implements the side information table by means of the PARTNER resource. Partner resources are defined by the CEDA DEFINE command. To become known to an active CICS system, a defined partner resource must then be installed using the CEDA INSTALL command.

Here is the general format of the CEDA DEFINE PARTNER command, which identifies a partner program in a remote LU:

```
CEDA DEFINE
   PARTNER(sym_dest_name)
   [GROUP(groupname)]
   [NETWORK(name)]
   NETNAME(name)
   [PROFILE(name)]
   {TPNAME(name)|XTPNAME(value)}
```

PARTNER(sym_dest_name)
The Initialize_Conversation (CMINIT) call identifies the partner program by this name. It must therefore be specified.

GROUP(groupname)
This identifies the group in the CICS system definition data set (CSD) of which the PARTNER definition is a member. Every CICS resource definition belongs to a group.

NETWORK(name)
This represents the network ID part of the *partner_LU_name*. CICS currently rules that LU names must be unique throughout all connected networks. This parameter is therefore ignored by CICS when processing an EXEC CICS ALLOCATE request. This parameter is included to support portability.

NETNAME(name)
This represents the network LU part of the *partner_LU_name*. It matches *name* in NETNAME(*name*) on a corresponding CEDA DEFINE CONNECTION command, which defines the partner LU in CICS.

PROFILE(name)
This specifies the name of the communications profile containing the mode name for this partner.

This parameter assigns a communication profile to the session. The name of this profile should match the *name* on a corresponding CEDA DEFINE

PROFILE(*name*) command. CICS communication profiles can contain the name of the group of LU 6.2 sessions from which the session is to be acquired (MODENAME on the CEDA DEFINE PROFILE command or *mode_name* in CPI Communications), thereby enabling a particular class of service to be selected.

TPNAME(name)|XTPNAME(value)

The *TP_name* (transaction identifier in CICS) can be defined using either the TPNAME or XTPNAME parameter. Use TPNAME when the characters shown in Table 12 on page 190 can be used. Use XTPNAME to specify the hexadecimal values for characters that CICS does not allow for the TPNAME parameter.

When the conversation is initiated by Allocate (CMALLC), the *TP_name* is sent to the remote LU. If the remote LU is another CICS/ESA system, the transaction definitions in the remote system are searched for a matching TPNAME or XTPNAME. If a match is not found, another search is made on the transaction identifiers defined by the CEDA DEFINE TRANSACTION(*name*) command, using the first four characters of the *TP_name*. Optionally, the global user exit XZCATT can be used. This will convert *TP_name*, which can be up to 64 characters in length, into a four-character CICS *tranid*.

Character Sets

Table 12 on page 190 gives details of character sets used in CICS to define the PARTNER operands. Where CICS names have to match with the variables listed in Table 7 on page 153, it is recommended that the character sets for CPI Communications, as defined in Table 12 on page 190, be used if portability is to be maintained.

Table 12. Defaults and Allowed Values for the CICS PARTNER Resource

Operand	Default	Mandatory	Type	Length	Range of values
PARTNER		yes	chars	8	A-Z 0-9
GROUP	current group	no	chars	1-8	A-Z 0-9 @#$ Lowercase changed to uppercase
NETWORK	undefined	no	chars	1-8	A-Z 0-9 @#$ Lowercase changed to uppercase
NETNAME	undefined	yes	chars	1-8	A-Z 0-9 @#$ Lowercase changed to uppercase
PROFILE	DFHCICSA[1]	no	chars	1-8	A-Z 0-9 ¢@#./-_%&$?!: |"=¬,;<>
TPNAME[2]	undefined	yes (or XTPNAME)	chars	1-64	A-Z 0-9 ¢@#./-_%&$?!: |"=¬,;<>
XTPNAME	undefined	yes (or TPNAME)	hex chars	2-128	0-9 A-F excluding the hex byte X'40'

Note:

1. The default profile DFHCICSA does not have a MODENAME defined.
2. TPNAME does not allow the characters a-z ( ) * + and ', which can be used in CPI Communications for *TP_name*. However, it is possible to specify these characters by giving their hexadecimal equivalents as XTPNAME.

Special CICS/ESA Notes

These are points that CICS programmers should note when writing programs for CPI Communications.

- A CICS transaction started by automatic transaction initiation (ATI) cannot use CPI Communications calls on its principal facility.
- On detecting error conditions that result in the return codes CM_PRODUCT_SPECIFIC_ERROR, CM_PARAMETER_ERROR, CM_PROGRAM_PARAMETER_CHECK, or CM_PROGRAM_STATE_CHECK, CICS sends an explanatory message to the CCPI transient data queue, which is used for logging CPI Communications messages.
- The CM_PRODUCT_SPECIFIC_ERROR return code results in one of the following informational error messages:
  - DFHCP0742 - the session is not available for CPI Communications as it is already in use by another process.
  - DFHCP0743 - CPI Communications cannot be used, as the transaction was initiated by ATI.
  - DFHCP0750 - an unrecognized profile name was supplied in partner resource *sym_dest_name*.

  There are no state transitions connected with CM_PRODUCT_SPECIFIC_ERROR.

- CICS indicates USER_ALREADY_VERIFIED in the FMH5 header for outgoing attach requests. The USER_ID is also included in the same header.
- The security requirements of incoming attach requests can be specified in CICS resource definition.
- CPI Communications applications in CICS cannot be SNA service programs and therefore cannot allocate on the mode name SNASVCMG. If they attempt to do this they will get CM_PARAMETER_ERROR.
- The *TP_name* may contain the CICS four-character transaction identifier of the partner program if the partner LU is CICS.
- CICS/ESA considers the scope of the *conversation_ID* to be one task.
- APPC transaction routing is supported for CPI Communications conversations.
- If CICS receives an unrecognized sense code in an FMH7 from the partner program, CICS will return either one of these return codes:

  ```
  CM_DEALLOCATED_ABEND
  CM_DEALLOCATED_ABEND_BO
  CM_DEALLOCATED_ABEND_SVC
  CM_DEALLOCATED_ABEND_SVC_BO
  CM_DEALLOCATED_ABEND_TIMER
  CM_DEALLOCATED_ABEND_TIMER_BO
  ```

  or one of these:

  ```
  CM_PROGRAM_ERROR_PURGING
  CM_SVC_ERROR_PURGING
  ```

  depending respectively on whether the FMH7 is accompanied by a conditional end bracket (CEB) or not. When this happens, CICS also sends an explanatory message containing the sense code received to the CCPI transient data queue.

  Note: The CEB is an SNA indicator used to say that the remote program deallocated the conversation.

- If a CICS transaction terminates while carrying on a conversation, CICS attempts to end the dangling conversation with a deallocate (CMDEAL) call. CICS first issues this call as if *deallocate_type* is set to CM_DEALLOCATE_FLUSH. However, if this call results in a state check, CICS reissues the CMDEAL call with *deallocate_type* set to CM_DEALLOCATE_ABEND.

  CICS provides this facility to allow emergency cleanup of sessions. The programmer should not rely on it when designing transactions. When this facility is used, information on the status of the conversation can be lost, because the conversation does not wait to receive the information. Instead, all conversations should be explicitly deallocated before the transaction is terminated.

Appendix F. CPI Communications on IMS/ESA

Programs running under IMS/ESA are able to use all the CPI Communications calls, extensions, and features provided by APPC/MVS. No special setup or restrictions apply. For a description of how to use CPI Communications under IMS/ESA, see Appendix G, "CPI Communications on MVS/ESA" on page 195 and the following IMS books:

- *IMS/ESA Application Programming: Data Communciation*
- *IMS/ESA Application Programming: DL/I Calls.*

Note: IMS application programs can use the CPI Communications pseudonym files provided by APPC/MVS as long as the APPC/MVS libraries containing these files are accessible when the programs are compiled.

Appendix G. CPI Communications on MVS/ESA

MVS application programs can use CPI Communications calls to communicate with programs on the same MVS system, other MVS systems, or other systems in an SNA network. The CPI Communications calls on MVS are compatible with those on other systems documented in this book, and are portable to other SAA systems.

This appendix describes aspects of CPI Communications that are unique to MVS. On MVS, programs that use CPI Communications calls are considered *transaction programs* (TPs). After reading this appendix, refer to *MVS/ESA Application Development: Writing Transaction Programs for APPC/MVS* for complete details on designing, writing, testing, and installing transaction programs to run on MVS. Refer to *MVS/ESA Planning: APPC Management* for details on how to supply side information on MVS, and how and when to create TP profiles for transaction programs that run on MVS.

MVS/ESA Operating Environment

MVS/ESA supports CPI Communications calls with the following distinctions:

- MVS/ESA does not log data associated with outgoing and incoming Send_Error and Deallocate (*Deallocate_type*= CM_DEALLOCATE_ABEND) calls.
- MVS/ESA does not support the Extract_Conversation_State call.
- MVS does not support a blank *sym_dest_name* value on the Initialize_Conversation call.
- The Test_Request_To_Send_Received (CMTRTS) call returns unpredictable results in conversations from MVS that cross a VTAM network. CMTRTS works properly in conversations within the same MVS system.
- MVS/ESA does not support conversations with a *sync_level* characteristic of CM_SYNC_POINT.
- The character set restrictions on LU names, VTAM mode names, and TP names differ slightly from those prescribed in Appendix A, "Variables and Characteristics" on page 147.

  LU names and mode names can contain uppercase alphabetic, numeric, and national characters ($, @, #), and must begin with an alphabetic or national character. IBM recommends that $, @, and # be avoided because they display differently depending on the national language code page in use.

  While TP names cannot contain blanks, blanks can still be used as a trailing pad character, but are not considered part of the string.

  IBM recommends that the asterisk (*) be avoided in MVS TP names because it causes a list request when entered on panels of the APPC administration dialog. The comma should also be avoided in MVS TP names because it acts as a parameter delimiter in DISPLAY APPC commands.

  See Appendix A in *MVS/ESA Application Development: Writing Transaction Programs for APPC/MVS* for a table showing the allowable characters.

Side Information

In MVS/ESA, you can supply the following side information for each *sym_dest_name* used on an Initialize_Conversation call:

- TP name
- Logon mode name
- Partner LU name.

Each *sym_dest_name* must have an entry containing this information in a side information file on MVS. The side information file is a VSAM key-sequenced data set. To add and maintain side information entries, your installation can use the APPC administration utility (ATBSDFMU) or the interactive APPC administration dialog that is provided with TSO/E Version 2 Release 3. For details about creating and maintaining side information on MVS, see *MVS/ESA Planning: APPC Management.*

TP Profiles

In MVS/ESA, program-startup processing uses a TP profile to start a local program in response to an allocation request from a remote program. The TP profile contains security and scheduling information for the local program, and must be created in advance using the APPC administration utility or APPC administration dialog. For details about the contents of TP profiles, and how to create and maintain them on MVS, see *MVS/ESA Planning: APPC Management.*

Invocation Details for CPI Communications

Pseudonym files for the CPI Communications calls in the following languages are shipped in SYS1.SAMPLIB with MVS/ESA and TSO/E.

MVS/ESA provides:

Language	**SYS1.SAMPLIB Member**
C	ATBCMC
COBOL	ATBCMCOB
FORTRAN	ATBCMFOR
PL/I	ATBCMPLI

Table 13. CPI Communications Pseudonym Files on MVS

All the above languages have an include or copy feature that allows the pseudonym files to reside in a single library and be used by multiple programs. These files can then be included or copied into programs that use CPI Communications calls.

TSO/E provides:

Language	**SYS1.SAMPLIB Member**
REXX	REXAPPC2

Table 14. CPI Communications Pseudonym Files on TSO/E

For REXX execs, copy the pseudonym file into the exec itself.

A CSP pseudonym file (CMCSP) is shipped with CSP 3.3.0 and above.

MVS Transaction Program Environment

Any MVS program that issues CPI Communications calls, or is attached by an APPC/MVS LU in response to an inbound request, is considered to be an APPC/MVS transaction program. To issue CPI Communication calls, a transaction program must meet the following requirements.

Requirements for TPs in Problem Program State

The following requirements apply to TPs that are written to run in problem program state (that is, using PSW key 8):

- CPI Communication calls must be invoked in 31-bit addressing mode.
- All parameters of CPI Communication calls must be addressable by the caller and in the primary address space.

If you are writing a TP to run in problem program state, you can skip over "General Requirements" and continue reading at "Linkage Conventions."

General Requirements

The following general requirements apply to CPI Communication calls invoked by any TP. They include requirements (such as cross memory allowed and locks not allowed) that are only of concern to TPs running in supervisor state or with PSW key 0-7.

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task or SRB mode
Cross memory mode:	PASN = HASN = SASN or PASN ¬= HASN ¬= SASN
Amode:	31-bit
ASC mode:	Primary or AR
Interrupt status:	Enabled for I/O and external interrupts
Locks:	No locks held
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Linkage Conventions

Any high-level language that conforms to the following linkage conventions may be used to issue CPI Communications calls on MVS.

- Register 1 must contain the address of a parameter list, which is a list of consecutive words, each containing the address of a parameter to be passed. The last word in this list must have a 1 in the high-order (sign) bit.
- Register 13 must contain the address of an 18-word save area.
- Register 14 must contain the return address.
- Register 15 must contain the entry point address of the service being called.
- If the caller is running in AR ASC mode, ARs 1, 13, 14, and 15 must all be set to zero.

On return from the service, general and access registers 2 through 14 are restored (registers 0, 1 and 15 are not restored).

Required Modules

Programs using CPI Communications on MVS, other than those written in REXX, must be link-edited with the load module ATBPBI, which is provided in SYS1.CSSLIB. TSO/E provides this service for REXX programs.

Scope of Conversation ID

MVS considers the scope of a transaction program to be the home address space. MVS allows programs to share a single conversation, represented by a *conversation_ID*, across multiple tasks or SRBs in the home address space. Passing of the *conversation_ID* between tasks and SRBs is the responsibility of the application; MVS does not provide a service for this purpose. Only one task or SRB can have control of the conversation at a given time, and only one CPI Communications call can be outstanding from one address space for a given conversation at a time. The only exception is the Deallocate call with a *deallocate_type* of CM_DEALLOCATE_ABEND, which can be issued from one task when a CPI Communications call is outstanding from another task for the same conversation. In that case, the outstanding call receives a return code of CM_PRODUCT_SPECIFIC_ERROR.

Product-Specific Errors in MVS/ESA

CPI Communications defines a return code called CM_PRODUCT_SPECIFIC_ERROR for each call. That return code may indicate one of several errors on MVS. For a list of possible product specific errors, see *MVS/ESA Application Development: Writing Transaction Programs for APPC/MVS*. On MVS, product-specific errors cause state transitions in the following situations:

- When a call is interrupted by a Deallocate call with *deallocate_type*= CM_DEALLOCATED_ABEND, the Deallocate call causes a state change to **Reset** state and the interrupted call receives a return code of CM_PRODUCT_SPECIFIC_ERROR.
- If APPC/MVS is deactivated, all active conversations are terminated; communicating programs receive CM_PRODUCT_SPECIFIC_ERROR in response to their next call and go into **Reset** state.

Deallocation of Dangling Conversations on MVS/ESA

Programs should deallocate conversations before ending, and should have recovery routines to deallocate conversations in the event of abnormal termination. If a program ends without deallocating its conversations, partner programs might continue to send data or wait to receive data. MVS deallocates any dangling conversations with a return code of CM_DEALLOCATED_ABEND_SVC.

MVS Performance Considerations

The relative performance speed of CPI Communications calls varies depending on the functions that the call performs. For example, calls that involve VTAM or cause the movement of data buffers involve a greater number of internal instructions. For an overview of performance considerations for CPI Communications calls on MVS, see *MVS/ESA Application Development: Writing Transaction Programs for APPC/MVS*.

APPC/MVS Services

In addition to CPI Communications calls, MVS also provides APPC/MVS callable services that are specific to MVS and provide similar and additional communications functions. For example, the APPC/MVS Send_Data (ATBSEND) call lets a program send data from an MVS/ESA data space. Many of the APPC/MVS calls also have an option to allow asynchronous execution. Programs on MVS can issue both CPI Communications (CMxxxx) and APPC/MVS (ATBxxxx) calls for the same conversation. The *conversation_ID* returned from a CMINIT or ATBALLC call is available to all subsequent CMxxxx or ATBxxxx calls for the conversation. For all conversation characteristics set by CMINIT, ATBALLC provides an equivalent parameter or sets the same default value. Calls to APPC/MVS services will not change any conversation characteristics previously established by a CPI Communications call. For more information about the APPC/MVS services, see *MVS/ESA Application Development: Writing Transaction Programs for APPC/MVS*.

Appendix H. CPI Communications on OS/2

This appendix provides information about the Operating System/2 (OS/2) implementation of CPI Communications. CPI Communications for OS/2 is provided as part of the IBM SAA Networking Services/2 Version 1.0 program product.[6] Throughout the remainder of this appendix, all references to Networking Services/2 refer to the IBM SAA Networking Services/2 Version 1.0 implementation of CPI Communications for OS/2.

The information in this appendix consists of:

- Defining and changing side information
- Additional Networking Services/2 calls
- Considerations for CPI Communications calls
- Variables and characteristics for Networking Services/2 calls
- CPI Communications functions not available on Networking Services/2
- Error return codes and error logging
- Defining and running a CPI Communications program on Networking Services/2
- Programming languages available for CPI Communications
- Pseudonym files for Networking Services/2 calls
- Sample programs for Networking Services/2

Networking Services/2 implements all of the functions in the CPI Communications element of Systems Application Architecture, except as described in "CPI Communications Functions Not Available" on page 230.

Defining and Changing Side Information

The set of parameters associated with a given symbolic destination name is called a side information entry. This section provides an overview of how an Networking Services/2 user or program can add, replace, delete, and extract side information entries. For information about all of the Networking Services/2 configuration capabilities, including details about how a user can configure CPI Communications side information, see *Networking Services/2 Installation and Network Administrator's Guide.*

When Networking Services/2 is started, it copies the side information from the active configuration file into internal memory. From then on, until it is restarted, Networking Services/2 maintains the side information within its internal memory. When a program calls Initialize_Conversation, Networking Services/2 obtains the initial values for the applicable conversation characteristics from this internal side information.

There are three ways to update Networking Services/2 CPI Communications side information: two by user control and one by program control.

[6] IBM SAA Networking Services/2 Version 1.0 provides advanced networking services for the Communications Manager of OS/2 Extended Edition Versions 1.2 and 1.3.

User-Defined Side Information

By using the Networking Services/2 configuration panels, the user is able to create and update side information entries in a configuration file. In addition, if the configuration file is active, the user may request that Networking Services/2 also update the side information in its internal memory. Of course, using the configuration panels requires an operator, a keyboard, and a display—a typical, if not universal, Networking Services/2 environment.

The user may also update the configuration file directly (for instance, by using an editor). This is useful, for example, when the user wants to configure side information for multiple systems, and then distribute the configuration file among the systems.

After updating the configuration file, the user may initiate Networking Services/2 verification of the configuration file. As an option, the user may request that changes made to the side information in the file also take effect in the internal side information, provided the configuration file is active and the verification is successful.

Program-Defined Side Information

A system management program may be written that issues Networking Services/2 calls to update the internal side information entries and obtain the parameter values of the entries. These updates affect only the Networking Services/2 internal side information, and remain in effect until changed; they do not alter the configuration file. Similarly, parameter values are obtained only from the internal side information; no reference is made to the configuration file. Consequently, these calls are useful for making temporary updates to the internal side information without affecting the original information in the configuration file.

Networking Services/2 provides two calls to update the internal side information and one to extract it. These are:

- Set_CPIC_Side_Information (XCMSSI)
  Add or replace the entry (all parameter values) for a symbolic destination name. See Page 219 for a detailed description.
- Delete_CPIC_Side_Information (XCMDSI)
  Delete the entry for a symbolic destination name. See Page 206 for a detailed description.
- Extract_CPIC_Side_Information (XCMESI)
  Return the entry for a symbolic destination name or for the *n*th entry. See Page 210 for a detailed description.

The symbolic destination name provides the index on each of these calls for accessing a side information entry. Alternatively, an integer value provides the index on the Extract_CPIC_Side_Information call for extracting an entry. When the program calls Set_CPIC_Side_Information and an entry for the symbolic destination name does not exist, a new entry is added. If the entry does exist, all of its parameters are replaced. When the program calls Delete_CPIC_Side_Information, the entire entry (comprising the symbolic destination name and all of its associated parameters) is removed. If the program calls Extract_CPIC_Side_Information, all parameters of the entry are returned except *security_password*.

The Communications Manager keylock feature is used with the Set_CPIC_Side_Information and Delete_CPIC_Side_Information calls. The feature may be enabled, or "secured," during Communications Manager configuration. It provides a means for protecting a system against unauthorized changes to Networking Services/2 system definition parameters, including CPI Communications side information. When the keylock feature has been secured, a program can issue the Set_CPIC_Side_Information and Delete_CPIC_Side_Information calls only if it supplies the Communications Manager master or service key. See *OS/2 Extended Edition Version 1.2 System Administrator's Guide for Communications* or *OS/2 Extended Edition Version 1.3 System Administrator's Guide for Communications* (referred to hereafter simply as the *System Administrator's Guide*) for details about the keylock feature.

Side Information Parameters

Table 15 shows the parameters contained in an entry of the Networking Services/2 CPI Communications side information, and a brief description of each parameter. Refer to "Characteristics, Fields, and Variables" on page 226 for a definition of the data type and length of these parameters.

Table 15 (Page 1 of 2). An Entry of Networking Services/2 CPI Communications Side Information

Parameter Pseudonym	Description
sym_dest_name	The name a program specifies on Initialize_Conversation to assign the initial characteristic values for the conversation.
partner_LU_name	The alias or network name of the partner LU for the conversation.
TP_name_type	An indicator of whether the *TP_name* parameter specifies an application TP name or an SNA service TP name. The value may be: • XC_APPLICATION_TP • XC_SNA_SERVICE_TP
TP_name	The name of the partner program for the conversation. The program may be an application TP or an SNA service TP.
mode_name	The name of the session mode for the conversation.
conversation_security_type	The level of access security information to include on the allocation request sent to the partner LU. This parameter is defined only from the configuration panels or by a program call to Set_CPIC_Side_Information; it is not included in the editable configuration file. The value may be: • XC_SECURITY_NONE • XC_SECURITY_SAME • XC_SECURITY_PROGRAM See "Set_Conversation_Security_Type (XCSCST)" on page 215 for a description of these values.

Table 15 (Page 2 of 2). An Entry of Networking Services/2 CPI Communications Side Information

Parameter Pseudonym	Description
security_user_ID	The access security user ID to include on the allocation request sent to the partner LU, when *conversation_security_type* is XC_SECURITY_PROGRAM. When *conversation_security_type* is other than XC_SECURITY_PROGRAM, this parameter is ignored. This parameter is defined only from the configuration panels or by a program call to Set_CPIC_Side_Information; it is not included in the editable configuration file.
security_password	The access security password to include on the allocation request sent to the partner LU, when *conversation_security_type* is XC_SECURITY_PROGRAM. When *conversation_security_type* is other than XC_SECURITY_PROGRAM, this parameter is ignored. This parameter is defined only from the configuration panels or by a program call to Set_CPIC_Side_Information; it is not included in the editable configuration file.

Networking Services/2 Calls

Networking Services/2 extends CPI Communications with additional calls a program may issue for system management and conversations.

The following two sections provide overviews of the system management and conversation calls. Following the overviews, all the calls are described in detail, in alphabetical order of their names.

Note: Using any of these calls means that the program will require modification to run on another SAA system that does not implement the call or implements it differently.

Overview of System Management Calls

These calls permit a program to change or obtain side information on Networking Services/2. A program issues these system management calls to set or delete parameter values in Networking Services/2 internal side information. The side information is used to assign initial characteristic values on the Initialize_Conversation (CMINIT) call. The program may also extract the current values of the side information.

Two of these calls include a *key* variable. The program must specify the master or service key when the Communications Manager keylock feature has been secured. The use of the *key* deters unintentional or unauthorized changes to the side information.

Table 16 lists the system management call names and gives a brief description of their function.

Table 16. List of Networking Services/2 System Management Calls for CPI Communications

Call	Pseudonym	Description
XCMDSI	Delete_CPIC_Side_Information	Deletes a side information entry for a specified symbolic destination name.
XCMESI	Extract_CPIC_Side_Information	Returns the parameter values of a side information entry for a specified symbolic destination name or entry number.
XCMSSI	Set_CPIC_Side_Information	Sets the parameter values of a side information entry for a specified symbolic destination name. If the entry does not exist in the side information, this call adds a new entry; otherwise, it replaces the existing entry in its entirety.

Overview of Networking Services/2 Conversation Calls

These calls permit a program to obtain or change the values for conversation-security characteristics available on Networking Services/2. As an aid to portability, where similar calls exist in other SAA environments, the same call names and syntaxes are used.

A program issues these calls to set the conversation security characteristics used with the Allocate (CMALLC) call. The program may also obtain the current values of these characteristics, except for *security_password*. This characteristic can be set, but it cannot be extracted; this restriction is intended to reduce the risk of unintentional or unauthorized access to passwords.

Table 17 lists the Networking Services/2 conversation calls and gives a brief description of their function.

Table 17. List of Networking Services/2 Conversation Calls for CPI Communications

Call	Pseudonym	Description
XCECST	Extract_Conversation_Security_Type	Returns the current value of the *conversation_security_type* characteristic.
XCECSU	Extract_Conversation_Security_User_ID	Returns the current value of the *security_user_ID* characteristic.
XCSCSP	Set_Conversation_Security_Password	Sets the value of the *security_password* characteristic.
XCSCST	Set_Conversation_Security_Type	Sets the value of the *conversation_security_type* characteristic.
XCSCSU	Set_Conversation_Security_User_ID	Sets the value of the *security_user_ID* characteristic.

Delete_CPIC_Side_Information (XCMDSI)

A program issues the Delete_CPIC_Side_Information (XCMDSI) call to delete an entry from Networking Services/2 internal side information. The entry to be deleted is identified by the symbolic destination name. Side information in the configuration file remains unchanged.

See "Defining and Changing Side Information" on page 201 for more information about configuring side information.

Format

```
CALL XCMDSI(key,
            sym_dest_name,
            return_code)
```

Parameters

key *(input)*
Specifies either the master or service key, when the Communications Manager keylock feature has been secured. See the *System Administrator's Guide* for details of the keylock feature.

sym_dest_name *(input)*
Specifies the symbolic destination name for the side information entry to be removed.

return_code *(output)*
Specifies the variable used to pass back the return code to the calling program. The *return_code* variable can have one of the following values:

- CM_OK
  Successful completion.
- CM_PROGRAM_PARAMETER_CHECK
  This return code indicates one of the following:
  - The *key* variable contains a value that does not match the master or service key and the Communications Manager keylock feature has been secured.
  - The *sym_dest_name* variable contains a name that does not exist in Networking Services/2 internal side information.
  - The address of one of the variables is invalid.
- CM_PRODUCT_SPECIFIC_ERROR
  The APPC component of Networking Services/2 is not active because of one of the following reasons:
  - The Communications Manager has not started APPC.
  - The Communications Manager has stopped APPC.
  - APPC is in an abended state.

State Changes

This call does not cause a state change on any conversation.

Usage Notes

1. If a *return_code* other than CM_OK is returned on this call, the side information entry is not deleted.
2. This call does not affect any active conversation.
3. The side information is removed immediately, which affects all Initialize_Conversation calls for the deleted symbolic destination name made after completion of this call.
4. While Networking Services/2 is removing the side information entry, any other program's call to change or extract the side information will be suspended until this call is completed; this includes a program's call to Initialize_Conversation (CMINIT).
5. The Delete_CPIC_Side_Information call is available for all SAA programming languages that Networking Services/2 supports.

Extract_Conversation_Security_Type (XCECST)

A program issues the Extract_Conversation_Security_Type (XCECST) call to obtain the access security type for the conversation.

Format

```
CALL XCECST(conversation_ID,
            conversation_security_type,
            return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

conversation_security_type *(output)*
Specifies the variable used to return the value of the *conversation_security_type* characteristic for this conversation. The *conversation_security_type* returned to the program can be one of the following:

- XC_SECURITY_NONE
- XC_SECURITY_SAME
- XC_SECURITY_PROGRAM

See "Set_Conversation_Security_Type (XCSCST)" on page 215 for a description of these values.

return_code *(output)*
Specifies the variable used to pass back the return code to the calling program. The *return_code* variable can have one of the following values:

- CM_OK
  Successful completion.
- CM_PROGRAM_PARAMETER_CHECK
  The *conversation_ID* specifies an unassigned conversation identifier.

State Changes

This call does not cause a state change.

Usage Notes

1. If a *return_code* other than CM_OK is returned on this call, the value contained in the *conversation_security_type* variable is not meaningful.
2. This call does not change the conversation security type for the specified conversation.
3. The *conversation_security_type* characteristic is set to an initial value from side information using the Initialize_Conversation (CMINIT) call. It can be set to a different value using the Set_Conversation_Security_Type (XCSCST) call.

Extract_Conversation_Security_User_ID (XCECSU)

A program issues the Extract_Conversation_Security_User_ID (XCECSU) call to obtain the access security user ID associated with a conversation.

Format

```
CALL XCECSU(conversation_ID,
            security_user_ID,
            security_user_ID_length,
            return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

security_user_ID *(output)*
Specifies the variable used to return the value of the *security_user_ID* characteristic for this conversation.

security_user_ID_length *(output)*
Specifies the variable used to return the length, in bytes, of the *security_user_ID* characteristic for this conversation.

return_code *(output)*
Specifies the variable used to pass back the return code to the calling program. The *return_code* variable can have one of the following values:

- CM_OK
  Successful completion.
- CM_PROGRAM_PARAMETER_CHECK
  The *conversation_ID* specifies an unassigned conversation identifier.

State Changes

This call does not cause a state change.

Usage Notes

1. If a *return_code* other than CM_OK is returned on this call, the values contained in the *security_user_ID* and *security_user_ID_length* variables are not meaningful.
2. The value returned in the *security_user_ID* variable is meaningful only for the length returned in the *security_user_ID_length* variable. Networking Services/2 does not fill out the *security_user_ID* variable with space characters.
3. This call does not change the security user ID for the specified conversation.
4. The *security_user_ID* characteristic is set to an initial value from side information using the Initialize_Conversation (CMINIT) call. It can be set to a different value using the Set_Conversation_Security_User_ID (XCSCSU) call.

Extract_CPIC_Side_Information (XCMESI)

A program issues the Extract_CPIC_Side_Information (XCMESI) call to obtain the parameter values of an entry in Networking Services/2 internal side information. The particular entry is requested by either an entry number or a symbolic destination name. Side information in the configuration file is not accessed.

See "Defining and Changing Side Information" on page 201 for information about configuring side information.

Format

```
CALL XCMESI(entry_number,
            sym_dest_name,
            side_info_entry,
            side_info_entry_length,
            return_code)
```

Parameters

entry_number *(input)*
Specifies the current number, or index, of the side information entry for which parameter values are to be returned, where an *entry_number* of 1 designates the first entry. The program may obtain parameter values from all the entries by incrementing the *entry_number* on successive calls until the last entry has been accessed; the program will get a *return_code* of CM_PROGRAM_PARAMETER_CHECK when the *entry_number* exceeds the number of entries in the side information.

Alternatively, the program may specify an *entry_number* of 0 to obtain a named entry, using the *sym_dest_name* variable to identify the entry.

sym_dest_name *(input)*
Specifies the symbolic destination name of the entry, when parameter values for a named entry are desired. Networking Services/2 uses this variable only when *entry_number* is 0. If *entry_number* is greater than 0, Networking Services/2 ignores this *sym_dest_name* variable.

side_info_entry *(output)*
Specifies a structure in which the parameter values are returned. The format of the structure is shown in Table 18. Values within character string fields are returned left-justified and filled on the right with space characters.

Table 18. Entry Structure for Extract_CPIC_Side_Information

Byte Offset	Field Length and Type	Parameter Pseudonym
0	8-byte character string	*sym_dest_name*
8	17-byte character string	*partner_LU_name*
25	3-byte character string	(reserved)
28	32-bit integer	*TP_name_type*
32	64-byte character string	*TP_name*
96	8-byte character string	*mode_name*
104	32-bit integer	*conversation_security_type*
108	8-byte character string	*security_user_ID*
116	8-byte character string	(reserved)

Refer to Table 15 on page 203 for a description of the side information parameters, and Table 24 on page 228 for a definition of the character set usage and length of each character string parameter.

side_info_entry_length *(input)*
Specifies the length of the entry structure. Set this to 124 (the only defined value).

return_code *(output)*
Specifies the variable used to pass back the return code to the calling program. The *return_code* variable can have one of the following values:

- CM_OK
  Successful completion.
- CM_PROGRAM_PARAMETER_CHECK
  This return code indicates one of the following:
  - The *entry_number* specifies a value greater than the number of entries in the side information.
  - The *entry_number* specifies a value less than 0.
  - The *sym_dest_name* specifies a name that is not in any entry in the internal side information, and *entry_number* specifies 0.
  - The *side_info_entry_length* contains a value other than 124.
  - The address of one of the variables is invalid.
- CM_PRODUCT_SPECIFIC_ERROR
  The APPC component of Networking Services/2 is not active because of one of the following reasons:
  - The Communications Manager has not started APPC.
  - The Communications Manager has stopped APPC.
  - APPC is in an abended state.

State Changes

This call does not cause a state change on any conversation.

Usage Notes

1. If a *return_code* other than CM_OK is returned on this call, the values contained in the *side_info_entry* fields are not meaningful.
2. If no user ID exists in the side information, the *security_user_ID* field will contain all space characters.
3. The *security_password* of the side information entry is not returned, and the field (at byte offset 116) is reserved on this call and its content is not meaningful.
4. This call does not affect any active conversation.
5. This call does not change the parameter values of the specified side information entry.
6. While Networking Services/2 is extracting the side information, any other program's call to change the side information will be suspended until this call is completed.
7. The Extract_CPIC_Side_Information call is available on Networking Services/2 for the following SAA programming languages:

   - C
   - COBOL
   - REXX

   Refer to "Programming Languages Available for CPI Communications" on page 238 for information on how to create the data structure using these languages.
8. The Extract_CPIC_Side_Information call and the Set_CPIC_Side_Information (XCMSSI) call use the same *side_info_entry* format. This enables a program to obtain an entry, update a field, and restore the updated entry, provided the entry contains no *security_password*, or the program also updates the *security_password*.
9. The *entry_number* specifies an index into the current list of internal side information entries. If entries are deleted, the indexes for particular entries may change.

Set_Conversation_Security_Password (XCSCSP)

A program issues the Set_Conversation_Security_Password (XCSCSP) call to set the access security password for a conversation. The password is necessary to establish a conversation on which a *conversation_security_type* of XC_SECURITY_PROGRAM is used. This call overrides the password that was assigned when the conversation was initialized.

Set_Conversation_Security_Password can be called only for a conversation that is in **Initialize** state and has a *conversation_security_type* of XC_SECURITY_PROGRAM.

Format

```
CALL XCSCSP(conversation_ID,
            security_password,
            security_password_length,
            return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

security_password *(input)*
Specifies the access security password. The partner LU uses this value and the *security_user_ID* to verify the identity of the requestor; the user ID can be specified by using the Set_Conversation_Security_User_ID (XCSCSU) call.

security_password_length *(input)*
Specifies the length of the security password. The length can be from 1 to 8 characters.

return_code *(output)*
Specifies the variable used to pass back the return code to the calling program. The *return_code* variable can have one of the following values:

- CM_OK
  Successful completion.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation ID.
  - The *security_password_length* specifies a value less than 1 or greater than 8.
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The conversation is not in **Initialize** state.
  - The *conversation_security_type* is not XC_SECURITY_PROGRAM.

State Changes

This call does not cause a state change.

Usage Notes

1. If a *return_code* other than CM_OK is returned on this call, the *security_password* characteristic remains unchanged.
2. When the program calls Initialize_Conversation (CMINIT), Networking Services/2 initializes the *security_password* to the value specified in the side information. A program should issue this Set_Conversation_Security_Password

call only if it desires a security password other than the password specified in the side information for this conversation.

3. When the program calls Allocate (CMALLC), and the *conversation_security_type* characteristic is XC_SECURITY_PROGRAM, Networking Services/2 obtains the access security password for the allocation request from the *security_password* characteristic. If the *conversation_security_type* is other than XC_SECURITY_PROGRAM, Networking Services/2 ignores the *security_password* characteristic when the program calls Allocate.

4. Specification of a security password that is invalid is not detected on this call. It is detected by the partner LU when it receives the allocation request. The partner LU returns an error indication to the local LU, which reports the error to the program by means of the CM_SECURITY_NOT_VALID return code on a subsequent call following the Allocate.

5. Specify *security_password* using the native encoding for Networking Services/2, which is 8-bit ASCII. Networking Services/2 converts the password to EBCDIC when it includes the password on the allocation request it sends to a partner LU.

Set_Conversation_Security_Type (XCSCST)

A program issues the Set_Conversation_Security_Type (XCSCST) call to set the security type for the conversation. This call overrides the security type that was assigned when the conversation was initialized.

Set_Conversation_Security_Type can be called only for a conversation that is in **Initialize** state.

Format

```
CALL XCSCST(conversation_ID,
            conversation_security_type,
            return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

conversation_security_type *(input)*
Specifies the level of access security information to be sent to the partner LU on an allocation request. The access security information, if present, consists of a user ID and password, or a user ID and an indication that the security information has already been verified. This parameter must be set to one of the following values:

- XC_SECURITY_NONE
  No access security information is to be included on the allocation request to the partner LU.
- XC_SECURITY_SAME
  The user ID (if any) on the inbound allocation request that started the local program, or the user ID (if any) obtained from the OS/2 EE User Profile Management facility when the program was started locally, is to be included on the allocation request sent to the partner LU, together with an "already verified" indication (if a user ID is sent).
- XC_SECURITY_PROGRAM
  The *security_user_ID* and *security_password* characteristics provide the access security information to be included on the allocation request sent to the partner LU. The partner LU uses this security information to verify the identity of the requestor.

return_code *(output)*
Specifies the variable used to pass back the return code to the calling program. The *return_code* variable can have one of the following values:

- CM_OK
  Successful completion.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation ID.
  - The *conversation_security_type* specifies an undefined value.
- CM_PROGRAM_STATE_CHECK
  The conversation is not in **Initialize** state.

State Changes

This call does not cause a state change.

Usage Notes

1. If a *return_code* other than CM_OK is returned on this call, the *conversation_security_type* characteristic remains unchanged.
2. When the program calls Initialize_Conversation (CMINIT), Networking Services/2 initializes the *conversation_security_type* to the value specified in the side information. Therefore, a program should issue the Set_Conversation_Security_Type call only if it wants a security type other than the initial type.
3. If the program sets *conversation_security_type* to XC_SECURITY_NONE or XC_SECURITY_SAME, Networking Services/2 will ignore the values of the *security_user_ID* and *security_password* characteristics when the program calls Allocate (CMALLC). If the program sets *conversation_security_type* to XC_SECURITY_PROGRAM, Networking Services/2 obtains the access security information from the *security_password* and *security_user_ID* characteristics. In particular:
   - If the security type is set to XC_SECURITY_NONE, the allocation request sent to the partner LU will contain no access security information.
   - If the security type is set to XC_SECURITY_SAME, the allocation request sent to the partner LU will contain the same level of access security information (either none or a user ID) as that used to start the program, provided the partner LU will accept already-verified access security information from the local LU. If the partner LU will not accept already-verified security information, the allocation request will contain no security information.
   - If the security type is set to XC_SECURITY_PROGRAM, the allocation request sent to the partner LU will contain the user ID and password from the corresponding characteristics, provided the partner LU will accept access security information from the local LU. If the partner LU will not accept access security information, the allocation request will contain no security information.
4. If the program sets *conversation_security_type* to XC_SECURITY_PROGRAM, it may then call Set_Conversation_Security_Password (XCSCSP) to set the *security_password*, and Set_Conversation_Security_User_ID (XCSCSU) to set the *security_user_ID*. If the program does not set the *security_user_ID* or *security_password*, the initial values will be used for these characteristics when the program calls Allocate. See Table 21 on page 224 for a list of the initial values for the security characteristics.

Set_Conversation_Security_User_ID (XCSCSU)

A program issues the Set_Conversation_Security_User_ID (XCSCSU) call to set the access security user ID for a conversation. The user ID is necessary to establish a conversation on which a *conversation_security_type* of XC_SECURITY_PROGRAM is used. This call overrides the user ID that was assigned when the conversation was initialized.

Set_Conversation_Security_User_ID can be called only for a conversation that is in **Initialize** state and has a *conversation_security_type* of XC_SECURITY_PROGRAM.

Format

```
CALL XCSCSU(conversation_ID,
            security_user_ID,
            security_user_ID_length,
            return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation ID.

security_user_ID *(input)*
Specifies the access security user ID. The partner LU uses this value along with the *security_password* to verify the identity of the requestor; the password can be specified by using the Set_Conversation_Security_Password (XCSCSP) call.

security_user_ID_length *(input)*
Specifies the length of the security user ID. The length can be from 1 to 8 characters.

return_code *(output)*
Specifies the variable used to pass back the return code to the calling program. The *return_code* variable can have one of the following values:

- CM_OK
  Successful completion.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation ID.
  - The *security_user_ID_length* specifies a value less than 1 or greater than 8.
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The conversation is not in **Initialize** state.
  - The *conversation_security_type* is not XC_SECURITY_PROGRAM.

State Changes

This call does not cause a state change.

Usage Notes

1. If a *return_code* other than CM_OK is returned on this call, the *security_user_ID* characteristic remains unchanged.

2. When the program calls Initialize_Conversation (CMINIT), Networking Services/2 initializes the *security_user_ID* to the value specified in the side information. A program should issue this Set_Conversation_Security_User_ID call only if it desires a security user ID other than the user ID specified in the side information for this conversation.

3. When the program calls Allocate (CMALLC), and the *conversation_security_type* characteristic is XC_SECURITY_PROGRAM, Networking Services/2 obtains the access security user ID for the allocation request from the *security_user_ID* characteristic. If the *conversation_security_type* is other than XC_SECURITY_PROGRAM, Networking Services/2 ignores the *security_user_ID* characteristic when the program calls Allocate.

4. Specification of a security user ID that is invalid is not detected on this call. It is detected by the partner LU when it receives the allocation request. The partner LU returns an error indication to the local LU, which reports the error to the program by means of the CM_SECURITY_NOT_VALID return code on a subsequent call following the Allocate.

5. Specify *security_user_ID* using the native encoding for Networking Services/2, which is 8-bit ASCII. Networking Services/2 converts the user ID to EBCDIC when it includes the user ID on the allocation request it sends to a partner LU.

Set_CPIC_Side_Information (XCMSSI)

A program issues the Set_CPIC_Side_Information (XCMSSI) call to add or replace an entry in Networking Services/2 internal side information. The entry contains all the side information parameters for the conversation identified by the supplied symbolic destination name. Side information in the configuration file remains unchanged. This call overrides the side information copied from the active configuration file when Networking Services/2 was started.

See "Defining and Changing Side Information" on page 201 for more information about configuring side information.

Format

```
CALL XCMSSI(key,
            side_info_entry,
            side_info_entry_length,
            return_code)
```

Parameters

key *(input)*
Specifies either the master or service key, when the Communications Manager keylock feature has been secured. See the *System Administrator's Guide* for details of the keylock feature.

side_info_entry *(input)*
Specifies the structure containing the parameter values for the side information entry. The format of the structure is shown in Table 19. Values within character string fields must be left-justified and filled on the right with space characters.

Table 19. Entry Structure for Set_CPIC_Side_Information

Byte Offset	Field Length and Type	Parameter Pseudonym
0	8-byte character string	*sym_dest_name*
8	17-byte character string	*partner_LU_name*
25	3-byte character string	(reserved)
28	32-bit integer	*TP_name_type*
32	64-byte character string	*TP_name*
96	8-byte character string	*mode_name*
104	32-bit integer	*conversation_security_type*
108	8-byte character string	*security_user_ID*
116	8-byte character string	*security_password*

Refer to Table 15 on page 203 for a description of the side information parameters, and Table 24 on page 228 for a definition of the character set usage and the length of each character string parameter.

side_info_entry_length *(input)*
Specifies the length of the entry structure. Set this to 124 (the only defined value).

return_code *(output)*
Specifies the variable used to pass back the return code to the calling program. The *return_code* variable can have one of the following values:

- CM_OK
  Successful completion.
- CM_PROGRAM_PARAMETER_CHECK
  This return code indicates one of the following:
  - The *key* variable contains a value that does not match the master or service key and the Communications Manager keylock feature has been secured.
  - The *sym_dest_name* field in the *side_info_entry* structure contains a space character in the leftmost byte (at byte offset 0 of the structure).
  - The *TP_name_type* field in the *side_info_entry* structure specifies an undefined value.
  - The *conversation_security_type* field in the *side_info_entry* structure specifies an undefined value.
  - The *side_info_entry_length* contains a value other than 124.
  - The address of one of the variables is invalid.
- CM_PRODUCT_SPECIFIC_ERROR
  The APPC component of Networking Services/2 is not active because of one of the following reasons:
  - The Communications Manager has not started APPC.
  - The Communications Manager has stopped APPC.
  - APPC is in an abended state.

State Changes

This call does not cause a state change on any conversation.

Usage Notes

1. If a *return_code* other than CM_OK is returned on this call, the side information entry is not created or changed.

2. The character string parameter values supplied in the fields of the *side_info_entry* structure are not checked for validity on this call. An invalid parameter value will be detected later on the Allocate (CMALLC) call or a subsequent call, such as Send_Data (CMSEND), depending on which parameter value is invalid. An invalid partner LU name or mode name will be detected on the Allocate call and indicated to the program on that call. An invalid TP name, user ID, or password will be detected by the partner LU when it receives the allocation request; in this case the partner LU returns an error indication, which is indicated to the program on a subsequent call following the Allocate.

3. This Set_CPIC_Side_Information call does not affect any active conversation.

4. The side information supplied on this call takes effect immediately and is used for all Initialize_Conversation calls for the new or changed symbolic destination name made after completion of this call.

5. While Networking Services/2 is updating the side information with the parameters from this call, any other program's call to change or extract the side information will be suspended until this call is completed; this includes a program's call to Initialize_Conversation (CMINIT).

6. The Set_CPIC_Side_Information call is available on Networking Services/2 for the following SAA programming languages:

   - C
   - COBOL
   - REXX.

   Refer to "Programming Languages Available for CPI Communications" on page 238 for information on how to create the data structure using these languages.

7. The Set_CPIC_Side_Information call and the Extract_CPIC_Side_Information (XCMESI) call use the same *side_info_entry* format. This enables a program to obtain an entry, update a field, and restore the updated entry, provided the entry contains no *security_password*, or the program also updates the *security_password*.

Considerations for CPI Communications Calls

This section describes CPI Communications calls that require special consideration when writing a CPI Communications program to be run on a Networking Services/2 system. Each call needing special attention is discussed in alphabetical order of the call name. Calls not included in this section need no special consideration.

Note: Explanations of error return codes whose causes on Networking Services/2 differ from those defined for CPI Communications are not included in this section. See "Error Return Codes and Error Logging" on page 232 for this information.

The CPI Communications Program and the Networking Services/2 Transaction Program Instance

When a program issues the Accept_Conversation (CMACCP) call or it first issues the Initialize_Conversation (CMINIT) call, Networking Services/2 creates an executable instance of a transaction program. From that point on, and until the program ends, the transaction program instance remains active. After the program issues the CMACCP or its first CMINIT call, all the CPI Communications calls it issues are for that transaction program instance.

Networking Services/2 represents the transaction program instance by means of a transaction program identifier, or TP_ID. It converts each CPI Communications call, other than the Extract and Set calls, to an APPC verb, and makes a call across its APPC interface to process the verb. Each APPC verb includes the TP_ID as a parameter. Networking Services/2 associates with each transaction program instance the logical unit of work identifier and access security information (if any) that it obtains when it starts the TP instance. It maintains the correlation of this information to the TP_ID until the TP instance ends—that is, until the CPI Communications program ends.

Accept_Conversation (CMACCP)

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

Table 20. Additional Characteristics Initialized Following CMACCP

Conversation Characteristic	Initialized Value
conversation_security_type	XC_SECURITY_SAME **Note:** This value is set regardless of the level of access security information (if any) on the inbound allocation request.
security_user_ID	A single space character.
security_user_ID_length	Set to 1.
security_password	A single space character.
security_password_length	Set to 1.

The CMACCP call starts a transaction program instance. The TP instance remains active until the CPI Communications program ends. Therefore, the program should not issue another CMACCP call after one has completed successfully, because the TP instance is already active; if it does, Networking Services/2 will reject the call with a *return_code* of CM_PRODUCT_SPECIFIC_ERROR.

If a program can have both inbound and outbound conversations, it must issue this CMACCP call before it issues any Initialize_Conversation (CMINIT) calls. Otherwise, if it issues the CMINIT call first, Networking Services/2 will start a TP instance for the program, and a CMACCP call issued after that will fail because a TP instance is already active for the program.

If an operator-started CPI Communications program issues the CMACCP call, either the operator or the program must set the TP name in the OS/2 environment variable named APPCTPN before the program issues the call. When an inbound allocation request arrives with this TP name specified, Networking Services/2 completes the call. The TP name set in the environment variable is made up of ASCII characters; therefore, it must be an application TP name, not an SNA service TP name. See "Unsupported TP Names" on page 230 for an explanation of this restriction.

Notes:

1. The use of the APPCTPN environment variable is required for an operator-started CPI Communications program that issues the Accept_Conversation call. The TP name set in the environment variable is case sensitive; that is, lowercase letters are not converted to uppercase.
2. For an attach manager-started program, Networking Services/2 sets the APPCTPN environment variable with the TP name from the inbound allocation request for the conversation when it starts the program. It then uses the TP name from the environment variable to complete the subsequent Accept_Conversation call from the program. Therefore, in order for Networking Services/2 to match the Accept_Conversation call with the inbound conversation, the attach manager-started program should not set the environment variable to a different TP name.
3. Networking Services/2 recognizes certain error conditions on the Accept_Conversation call that are specific to its use of OS/2 environment variables. See "Error Return Codes and Error Logging" on page 232 for more details.

Allocate (CMALLC)

Networking Services/2 does not allow a CPI Communications program to allocate a conversation that uses the CPSVCMG or SNASVCMG mode name. If the program attempts to call Allocate (CMALLC) with the *mode_name* characteristic set to CPSVCMG or SNASVCMG, Networking Services/2 will reject the call with a *return_code* of CM_PARAMETER_ERROR.

Networking Services/2 does not recognize TP names made up of characters from a double-byte character set, such as Kanji. If the program calls Allocate (CMALLC) with the *TP_name* characteristic set to a double-byte name, Networking Services/2 will treat the name as ASCII and translate each byte to EBCDIC. The resulting TP name will be invalid, and the partner LU for the conversation will reject the allocation request.

Initialize_Conversation (CMINIT)

When Initialize_Conversation completes successfully, the following additional characteristics are initialized:

Table 21. Additional Characteristics Initialized Following CMINIT

Conversation Characteristic	**Initialized Value**
conversation_security_type	Security type from side information referenced by *sym_dest_name*. If a blank *sym_dest_name* is specified, *conversation_security_type* is set to XC_SECURITY_SAME.
security_user_ID	User ID from side information referenced by *sym_dest_name*, if *conversation_security_type* is XC_SECURITY_PROGRAM; otherwise, a single space character. If a blank *sym_dest_name* is specified, *security_user_ID* is set to a single space character.
security_user_ID_length	Length of security user ID, if *conversation_security_type* is XC_SECURITY_PROGRAM; otherwise, 1. If a blank *sym_dest_name* is specified, *security_user_ID_length* is set to 1.
security_password	Password from side information referenced by *sym_dest_name* if *conversation_security_type* is XC_SECURITY_PROGRAM; otherwise, a single space character. If a blank *sym_dest_name* is specified, *security_password* is set to a single space character.
security_password_length	Length of security password, if *conversation_security_type* is XC_SECURITY_PROGRAM; otherwise, 1. If a blank *sym_dest_name* is specified, *security_password_length* is set to 1.

The CMINIT call starts a transaction program instance if one is not already active for the CPI Communications program. The TP instance remains active until the program ends.

If an operator-started CPI Communications program is to be run on a local LU other than the default LU configured for the node, either the operator or the program must set the local LU name in an OS/2 environment variable named APPCLLU before the program issues the CMINIT call.

Notes:

1. The APPCLLU environment variable is used only for an operator-started CPI Communications program that issues the Initialize_Conversation call to start the TP instance. The LU name set in the environment variable is case sensitive; that is, lowercase letters are not converted to uppercase.
2. The APPCTPN environment variable is used to obtain the local TP name for any type CPI Communications program (operator-started or attach manager-started) that issues the Initialize_Conversation call to start the TP instance. Networking Services/2 sets the environment variable to a default local TP name of CPIC_DEFAULT_TPNAME. The operator or program may set the APPCTPN environment variable to a different TP name before the program issues the Initialize_Conversation call, if a different local TP name is desired. Networking Services/2 does not send this local TP name outside the node.

3. Networking Services/2 recognizes certain error conditions on the Initialize_Conversation call that are specific to its use of OS/2 environment variables. See "Error Return Codes and Error Logging" on page 232 for more details.

Receive (CMRCV)

When this call is receiving data from a basic conversation, the 2-byte logical record length, or LL, field of the data is in System/370 format, with the left byte being the most significant. Depending on the programming language used, the program may have to reverse the bytes in order to use the field value in an integer operation.

Networking Services/2 does not perform any EBCDIC to ASCII translation on the data before it is placed in the *buffer* variable.

Send_Data (CMSEND)

When this call is sending data on a basic conversation, the 2-byte logical record length, or LL, field of the data must be in System/370 format, with the left byte being the most significant. If the program obtains this value from an integer operation, it may have to reverse the bytes before issuing the call, depending on the programming language used.

Networking Services/2 does not perform any ASCII to EBCDIC translation on the data when it sends the data from the *buffer* variable.

Set_Log_Data (CMSLD)

The program sets the *log_data* characteristic to a character string representing the message text portion of the Error Log GDS variable. The *log_data* variable on the CMSLD call must contain a 1-512 byte ASCII character string, and it may include the space character. Networking Services/2 translates all *log_data* characters to EBCDIC before recording the log data in the system error log and sending the Error Log GDS variable on the conversation. The log data is written to the error log with a type of 0001 and a subtype set to the sense data Networking Services/2 sends with the Error Log GDS variable on the conversation.

Set_Mode_Name (CMSMN)

The program should not set the *mode_name* conversation characteristic to CPSVCMG nor SNASVCMG. Although Networking Services/2 allows the program to specify these mode names on the Set_Mode_Name call, it will reject the subsequent Allocate (CMALLC) call with a *return_code* of CM_PARAMETER_ERROR.

Set_Partner_LU_Name (CMSPLN)

The program can set the *partner_LU_name* characteristic to either an alias or network name. Alias and network names are distinguished from each other on this call as follows:

- An alias name is 1-8 characters and does not contain the period. Networking Services/2 retains alias names in ASCII, without translating them to EBCDIC.
- A network name is 2-17 characters, with the period separating the network ID (0-8 characters) and the network LU name (1-8 characters). If the network name does not include a network ID, the period must stil! be specified preceding the network LU name, to distinguish the name as a network name instead of an alias name.

Set_Sync_Level (CMSSL)

Networking Services/2 supports the following synchronization levels:

- CM_NONE
- CM_CONFIRM

If the program specifies CM_SYNC_POINT on the *sync_level* variable, Networking Services/2 will reject the Set_Sync_Level call with a *return_code* of CM_PROGRAM_PARAMETER_CHECK.

Set_TP_Name (CMSTPN)

The program can set the *TP_name* characteristic to an application TP name only. It cannot set the *TP_name* to an SNA service TP name, because the encoding of the first character of an SNA service TP name overlaps with, and is indistinguishable from, some ASCII characters. In order to allocate a conversation with a partner SNA service TP, the SNA service TP name must be defined in the side information entry for the conversation.

Characteristics, Fields, and Variables

This section defines the values and data types for the additional characteristics, fields, and variables used with the Networking Services/2 calls. It also includes the CPI Communications variables for which Networking Services/2 imposes certain restrictions.

The following distinction is made regarding characteristics, fields, and variables:

Characteristic
An internal parameter of a given conversation whose value is maintained within the CPI Communications component. The value of a conversation characteristic is initialized during the Initialize_Conversation (CMINIT) or Accept_Conversation (CMACCP) call for that conversation. The value may be subsequently changed using a Set call.

Field
An element of a data structure. The data structure itself is specified as a variable on the Set_CPIC_Side_Information and Extract_CPIC_Side_Information calls. A field can supply a value as input on a call, or return a value as output from a call.

Variable
A parameter specified on a call. It can supply a value as input on a call, or return a value as output from a call.

Note: Networking Services/2 does not support all values of the conversation characteristics and variables described in Appendix A, "Variables and Characteristics." The ones it does not support are listed in Table 25 on page 231.

Networking Services/2 Native Encoding

The native encoding for specifying certain character string variables and fields on Networking Services/2 is ASCII. These variables and fields are:

- *key*
- *mode_name**
- *partner_LU_name* (as an alias name)
- *partner_LU_name* (as a network name)*

- *security_password**
- *security_user_ID**
- *sym_dest_name*
- *TP name* (as an application TP name)*
- *TP name* (as an SNA service TP name, excluding first character).*

The variables and fields indicated by an asterisk (*) are translated from ASCII to EBCDIC on input, and from EBCDIC to ASCII on output. The others are retained in ASCII.

Networking Services/2 performs translation between the ASCII characters having code points in the range X'20' through X'7E' and the corresponding EBCDIC characters. The ASCII characters are those defined in the ASCII code page 850; however, this range of characters is common across all ASCII code pages. The EBCDIC characters are those defined in the EBCDIC code page 500.

The characters Networking Services/2 translates are all the characters from character set 00640 (including the space character) and the 13 additional characters listed in Table 22. Characters outside character set 00640 and not shown in this table are not translated and remain unchanged.

Table 22. Additional Characters Translated Between ASCII and EBCDIC

Graphic	Description
[	Left bracket
!	Exclamation point
]	Right bracket
$	Dollar sign
^	Caret
`	Right prime
#	Number sign
@	At sign
~	Tilde
|	Vertical bar
{	Left brace
}	Right brace
\	Back slash

Pseudonyms and Their Corresponding Integer Values

Integer characteristics, variables, and fields are shown throughout this appendix as having pseudonym values rather than integer values. For example, instead of stating that the variable *conversation_security_type* is set to an integer value of 0, this appendix shows *conversation_security_type* being set to the pseudonym value of XC_SECURITY_NONE.

Table 23 shows the mapping from pseudonyms to integer values for the additional characteristics, fields, and variables that Networking Services/2 provides.

Table 23. Variables, Characteristics and Fields, and Their Possible Values

Characteristic, Field, or Variable Name	Pseudonym Values	Integer Values
conversation_security_type	XC_SECURITY_NONE	0
	XC_SECURITY_SAME	1
	XC_SECURITY_PROGRAM	2
TP_name_type	XC_APPLICATION_TP	0
	XC_SNA_SERVICE_TP	1

Field and Variable Types and Lengths

Table 24 defines the data type, character set usage, and length of fields and variables used for the additional Networking Services/2 calls. It also includes CPI Communications variables for which Networking Services/2 imposes certain restrictions.

The variables Networking Services/2 uses for its conversation calls are of the same two types as CPI Communications uses: integer and character. The variables Networking Services/2 uses for its system management calls include a third type, a data structure. The data structure has both integer and character string fields.

With one exception, the program specifies the character string fields and variables using ASCII characters on the Set calls, and Networking Services/2 returns them as ASCII characters on the Extract calls. The one exception is the SNA service TP name, as noted in Table 24.

All fields of the *side_info_entry* structure are fixed length, so character string data within these fields must be specified, and is returned, left justified and filled on the right with ASCII space characters. In general, Networking Services/2 does not reject the Set_CPIC_Side_Information call when a character string field of the *side_info_entry* contains all space characters, even if the minimum length defined for the data in that field is greater than 0. The *sym_dest_name* is the exception, as noted in Table 24.

The use of character set 01134 or 00640, as defined for each field or variable, is recommended for consistency with the CPI Communications definition; however, Networking Services/2 does not enforce this.

Table 24 (Page 1 of 2). Variable and Field Types and Lengths

Variable or Field	Data Type	Character Set	Length
conversation_security_type	Integer	Not applicable	32 bits
key [1, 10]	Character string	01134	8 bytes
mode_name [2]	Character string	01134	0-8 bytes
partner_LU_name [3, 5] (as an alias name)	Character string	00640	1-8 bytes
partner_LU_name [4] (as a network name)	Character string	01134	2-17 bytes
security_password [5]	Character string	00640	1-8 bytes

Table 24 (Page 2 of 2). Variable and Field Types and Lengths

Variable or Field	Data Type	Character Set	Length
security_password_length	Integer	Not applicable	32 bits
security_user_ID [5]	Character string	00640	1-8 bytes
security_user_ID_length	Integer	Not applicable	32 bits
side_info_entry [6]	Data structure	Field dependent	124 bytes
side_info_entry_length	Integer	Not applicable	32 bits
sym_dest_name [7,11]	Character string	01134	8 bytes
TP_name [5,8] (as an application TP name)	Character string	00640	1-64 bytes
TP_name [9] (as an SNA service TP name)	Character string	01134	1-4 bytes
TP_name_type	Integer	Not applicable	32 bits

Referenced Notes:

1. The special characters @, #, and $ are also allowed as part of the key.
2. The null string (all space characters) is a valid mode name. The program should not set the *mode_name* to CPSVCMG or SNASVCMG. Although Networking Services/2 allows the program to specify these mode names, it will reject a program's Allocate (CMALLC) call with a *return_code* of CM_PARAMETER_ERROR if the conversation characteristic is set to either of these mode names.
3. The period character is not allowed as part of an alias partner LU name. An alias partner LU name may contain ASCII characters in the range X'21' to X'FE'; however, use of characters drawn from character set 00640 is recommended.
4. The period must be present in a network partner LU name because it distinguishes the name as a network name instead of an alias name. If the partner LU name does not have a network ID, the period must be the first character in the *partner_LU_name* variable or field.
5. The space character is not allowed as part of a partner LU name, security password, security user ID, or application TP name, as it is used as the fill character in the corresponding fields of the *side_info_entry* data structure.
6. The format of the *side_info_entry* data structure is shown in the description of "Set_CPIC_Side_Information (XCMSSI)" on page 219.
7. The *sym_dest_name* can be specified as all space characters only on the Initialize_Conversation (CMINIT) call. On all other calls that include this variable or field, the name must be 1-8 characters long.
8. An application TP name is composed entirely of ASCII characters. It cannot be a double-byte TP name—one that has a leading X'0E' byte and a trailing X'0F' byte—as Networking Services/2 does not support double-byte TP names. Networking Services/2 converts all characters of an application TP name from ASCII to EBCDIC when it includes the TP name on an allocation request.
9. An SNA service TP name is composed of a leading SNA service TP identifier byte and 0-3 additional ASCII characters; the identifier byte has a value in the range X'00' to X'0D' and X'0F' to X'3F'. An SNA service TP name may be

specified only with the Set_CPIC_Side_Information call; it cannot be specified on the Set_TP_Name call.

10. The *key* variable on the Delete_CPIC_Side_Information (XCMDSI) and Set_CPIC_Side_Information (XCMSSI) calls must be at least 8 bytes long. The key within the variable may be from 1 to 8 characters long. If the key is shorter than 8 characters, it must be left-justified in the variable and filled on the right with space characters. If the variable is longer than 8 bytes, the key is taken from the first (leftmost) 8 bytes and the remaining bytes are ignored.

11. The *sym_dest_name* variable on the Delete_CPIC_Side_Information (XCMDSI) and Extract_CPIC_Side_Information (XCMESI) calls must be at least 8 bytes long. The symbolic destination name within the variable may be from 1 to 8 characters long on these calls. If the symbolic destination name is shorter than 8 characters, it must be left-justified in the variable and filled on the right with space characters. If the variable is longer than 8 bytes, the symbolic destination name is taken from the first (leftmost) 8 bytes and the remaining bytes are ignored.

CPI Communications Functions Not Available

This section lists the CPI Communications functions that are not available at the CPI Communications interface on Networking Services/2.

Unsupported TP Names

Networking Services/2 does not support double-byte TP names, and provides limited support for SNA service TP names.

Double-Byte TP Names

Networking Services/2 does not support TP names made up of characters from a double-byte character set, such as Kanji. These TP names begin with the X'0E' character and end with the X'0F' character, and they have an even number of bytes (2 or more) between these delimiting characters.

If the program calls Allocate (CMALLC) with the *TP_name* characteristic set to a double-byte name, Networking Services/2 treats the name as ASCII and translates each byte to EBCDIC. The resulting TP name is invalid, and the partner LU for the conversation rejects the allocation request.

SNA Service TP Names

Networking Services/2 does not support the specification of an SNA service TP name on the Set_TP_Name (CMSTPN) call, nor does it support the setting of the APPCTPN OS/2 environment variable to an SNA service TP name.

Specifically, a TP name can be supplied in three ways:

- Specified on the Set_TP_Name (CMSTPN) call
- Specified in the side information—by means of either the configuration panels or the Set_CPIC_Side_Information (XCMSSI) call
- Set in the APPCTPN OS/2 environment variable for operator-started programs

A TP name specified on the CMSTPN call or in the side information is used when a program issues the Allocate (CMALLC) call for an outbound conversation. A TP name set in the APPCTPN OS/2 environment variable is used when an operator-started program issues the Accept_Conversation (CMACCP) call for an inbound conversation. Networking Services/2 treats a TP name specified on the

CMSTPN call or a TP name set in the APPCTPN OS/2 environment variable as an application TP name and translates all characters of the name from ASCII to EBCDIC before using the name.

However, Networking Services/2 is able to distinguish an application TP name from an SNA service TP name specified in the side information. The side information includes a flag, *TP_name_type*, that indicates whether the TP name is an application TP name or an SNA service TP name. Networking Services/2 translates all characters of an SNA service TP name, except the first character, from ASCII to EBCDIC before using the name.

Remotely started CPI Communications programs—local programs that are started by inbound allocation requests—cannot be SNA service TPs. That is, Networking Services/2 cannot complete an Accept_Conversation (CMACCP) call for an inbound allocation request that specifies an SNA service TP name. This restriction exists because Networking Services/2 sets the TP name (converted from EBCDIC to ASCII) in the APPCTPN OS/2 environment variable for the program. Then, when the program issues the CMACCP call, Networking Services/2 retrieves the TP name from the environment variable to complete the call, just as it does for an operator-started program. However, Networking Services/2 sets only application TP names in the environment variable, not SNA service TP names (because of the conflict with ASCII characters and the first character of an SNA service TP name). If the inbound allocation request specifies an SNA service TP name, the CPI Communications program's CMACCP will fail with a *return_code* of CM_PRODUCT_SPECIFIC_ERROR.

Unsupported Characteristic and Variable Values

Networking Services/2 does not support the synchronization level of CM_SYNC_POINT. Therefore, the values for conversation characteristics and variables listed in Table 25 are not supported on Networking Services/2. Networking Services/2 supports all other characteristic and variable values described in Chapter 4, "Call Reference Section."

Table 25 (Page 1 of 2). Variable and Characteristic Values Not Supported

Variable or Characteristic	Pseudonym Values	Integer Values
conversation_state	CM_DEFER_RECEIVE_STATE	9
	CM_DEFER_DEALLOCATE_STATE	10
	CM_SYNC_POINT_STATE	11
	CM_SYNC_POINT_SEND_STATE	12
	CM_SYNC_POINT_DEALLOCATE_STATE	13

Table 25 (Page 2 of 2). Variable and Characteristic Values Not Supported

Variable or Characteristic	Pseudonym Values	Integer Values
return_code	CM_SYNC_LEVEL_NOT_SUPPORTED_LU	7
	CM_TAKE_BACKOUT	100
	CM_DEALLOCATED_ABEND_BO	130
	CM_DEALLOCATED_ABEND_SVC_BO	131
	CM_DEALLOCATED_ABEND_TIMER_BO	132
	CM_RESOURCE_FAIL_NO_RETRY_BO	133
	CM_RESOURCE_FAILURE_RETRY_BO	134
	CM_DEALLOCATED_NORMAL_BO	135
status_received	CM_TAKE_COMMIT	5
	CM_TAKE_COMMIT_SEND	6
	CM_TAKE_COMMIT_DEALLOCATE	7
sync_level	CM_SYNC_POINT	2

Error Return Codes and Error Logging

This section discusses error return codes on Networking Services/2, but only for causes that differ from what is described in Appendix B, "Return Codes." For most CPI Communications error return codes, the causes on Networking Services/2 are the same. This section also discusses the causes for indicating the CM_PRODUCT_SPECIFIC_ERROR return code on Networking Services/2.

Networking Services/2 records (or logs) errors in the Communications Manager system error log that are associated with certain CPI Communications error return codes, and it logs all errors associated with the CM_PRODUCT_SPECIFIC_ERROR return code. Refer to *Networking Services/2 Problem Determination Guide* for a description of the error log entries.

Logging Errors for CPI Communications Error Return Codes

Networking Services/2 will record an entry in the Communications Manager system error log for certain errors for which CPI Communications has defined error return codes. In particular, Networking Services/2 logs session activation failures and session outages. If the CPI Communications program receives one of the following error return codes, a system error log entry may also have been recorded:

- CM_ALLOCATE_FAILURE_NO_RETRY
- CM_ALLOCATE_FAILURE_RETRY
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY

The first two return codes are indicated on an Allocate (CMALLC) call when Networking Services/2 fails to allocate a session. The last two return codes are indicated on calls following an Allocate or Accept_Conversation (CMACCP) when Networking Services/2 detects a session outage. Networking Services/2 logs session outages in all cases. However, it logs allocation failures only when the cause is a session activation failure. Allocation failures caused by the session limit being 0 for the session mode name are not logged.

Session activation errors and session outages can be caused by a number of different conditions. The system error log specifies a type and subtype combination for each condition.

Networking Services/2 also logs the error log data sent by the partner program. These error log entries all have a type of 0001, and a subtype set to the sense data Networking Services/2 receives with the error notification. The CPI Communications return codes for these conditions are:

- CM_DEALLOCATE_ABEND
- CM_PROGRAM_ERROR_NO_TRUNC
- CM_PROGRAM_ERROR_PURGING
- CM_PROGRAM_ERROR_TRUNC
- CM_SVC_ERROR_NO_TRUNC
- CM_SVC_ERROR_PURGING
- CM_SVC_ERROR_TRUNC

Causes for the CM_PROGRAM_PARAMETER_CHECK Return Code

This section discusses the causes for the CM_PROGRAM_PARAMETER_CHECK return code that are specific to Networking Services/2. These causes are in addition to those described in Appendix B, "Return Codes." Networking Services/2 does not log these errors.

Networking Services/2 indicates the CM_PROGRAM_PARAMETER_CHECK return code for the following reasons:

- The call passed a pointer to a variable and the pointer is invalid. An invalid pointer contains a memory address that Networking Services/2 cannot use to refer to the variable. Examples of an invalid pointer are:
  - An address within a code segment
  - An address greater than the data segment limit
  - An address of zero (which is called a null address).
- The call passed a pointer to a character string variable (including a buffer variable) and a length, but the length would extend the valid addressability for the data beyond the segment limit.

The state of the conversation remains unchanged.

Causes for the CM_PROGRAM_STATE_CHECK Return Code

This section discusses the causes for the CM_PROGRAM_STATE_CHECK return code on the Accept_Conversation (CMACCP) call that are specific to Networking Services/2. These causes are in addition to what is described for the call. Networking Services/2 does not log these errors.

Networking Services/2 indicates the CM_PROGRAM_STATE_CHECK return code on the Accept_Conversation call for the following reasons:

- The operator or program set a TP name in the APPCTPN environment variable that was incorrect; that is, it did not match the TP name on the inbound allocation request for the program.
- An operator-started program issued the Accept_Conversation call, but the call expired before the inbound allocation request arrived. The duration that a call

waits for an inbound allocation request is configured on the TP definition, using the receive_allocate_timeout parameter.

- An operator-started or attach manager-started program issued the Accept_Conversation call after the inbound allocation request for the program expired. The duration that an inbound allocation request waits for an Accept_Conversation call is configured on the TP definition, using the incoming_allocate_timeout parameter.

Causes for the CM_PRODUCT_SPECIFIC_ERROR Return Code

This section lists the causes for indicating CM_PRODUCT_SPECIFIC_ERROR. Table 26 provides a brief description of each error for which the program can receive this return code. The table also lists the error log types and subtypes logged for each of these errors. Except as noted in the table, these errors apply across most calls, so they are not singled out on a call basis. The CPI Communications component records all these errors in the Communications Manager system error log. CPIC is indicated as the originator of each error log entry. Refer to *Networking Services/2 Problem Determination Guide* for a complete description of these errors and the recommended action to take.

Notes:

1. The state of the conversation remains unchanged, except for conditions where APPC has been stopped or has abnormally ended.
2. As part of its processing of a CPI Communications call other than a Set or Extract call, the CPI Communications component of Networking Services/2 creates an APPC verb and calls the APPC component to execute the verb. For example, when a program calls Allocate (CMALLC) for a basic conversation, Networking Services/2 creates and executes the APPC verb, ALLOCATE. When the APPC component of Networking Services/2 encounters an unexpected error on an OS/2 call, it logs the error as type 0022. Then, when the APPC component returns to the CPI Communications component, the latter logs an additional entry as type 0052. The second log entry indicates the error occurred while processing a CPI Communications call.
3. If the Communications Manager is not active when the program issues the Initialize_Conversation or Accept_Conversation call, this return code is returned on the call but no error is logged.

Table 26 (Page 1 of 2). Causes of the CM_PRODUCT_SPECIFIC_ERROR *Return Code*

Causes for CM_PRODUCT_SPECIFIC_ERROR	**Error Log Type**	**Error Log Subtype**
While processing a CPI Communications call, Networking Services/2 encountered an unexpected error on an OS/2 call it issued, and the error occurred within its CPI Communications component. The right half of the error log subtype contains the OS/2 request code. **Note:** Contrast this error with error log type 0052, subtype 00000002.	**0022**	**0000xxxx**

Table 26 (Page 2 of 2). Causes of the CM_PRODUCT_SPECIFIC_ERROR *Return Code*

Causes for CM_PRODUCT_SPECIFIC_ERROR	**Error Log Type**	**Error Log Subtype**
The program issued a call passing a pointer to a variable and the pointer is invalid. This error is logged only for the Accept_Conversation (CMACCP) call. **Note:** If the invalid pointer is for the *return_code* variable, Networking Services/2 cannot return from the call, and the program is terminated.	**0052**	**00000001**
As part of its processing of a CPI Communications call, the CPI Communications component of Networking Services/2 called the APPC component with an APPC verb and received one of the following error indications back from that call: • The APPC component is in an abended state. • The Communications Manager has stopped the APPC component. • The program's stack size is too small. • Networking Services/2 is currently processing a call for this program; this error can occur only if the program is running concurrently on multiple OS/2 threads. • The APPC component encountered an unexpected error on one of its OS/2 calls. • The program issued an Accept_Conversation (CMACCP) call and the APPC Attach Manager is stopped. • The program issued an Initialize_Conversation (CMINIT) call and the local LU name set in the APPCLLU OS/2 environment variable is not defined to Networking Services/2.	**0052**	**00000002**
The CPI Communications program issued an Accept_Conversation (CMACCP) call after the transaction program instance for the program was started. Networking Services/2 starts a TP instance for the program when it successfully completes a CMACCP call or the program's first Initialize_Conversation (CMINIT) call, whichever is first. The TP instance remains active until the program ends.	**0052**	**00000003**
The Communications Manager has not started the APPC component.	**0052**	**00000004**
The CPI Communications component has insufficient memory to satisfy the call. The maximum amount of memory available to the CPI Communications component is 655,360 bytes (10 maximum-size segments).	**0052**	**00000005**
An operator-started program issued an Accept_Conversation (CMACCP) call, but no TP name was set in the APPCTPN OS/2 environment variable.	**0052**	**00000006**

Defining and Running a CPI Communications Program on Networking Services/2

This section discusses the operating parameters that may be configured for a CPI Communications program and how they affect the operation of the program.

Defining a CPI Communications Program to Networking Services/2

A CPI Communications program is defined to Networking Services/2 by configuring a transaction program definition. The TP definition includes the OS/2 file specification (the drive, path, filename, and extension for the file containing the program), a TP name, and certain operating characteristics for the program. A program that is to be started by an inbound allocation request should be defined to Networking Services/2 as non-queued, attach manager started. This definition allows multiple instances of the program to be started concurrently when Networking Services/2 receives multiple inbound allocation requests for the TP name defined for the program.

The TP name must be an application TP name. A local CPI Communications program cannot be an SNA service TP—that is, the program cannot accept a conversation started by an inbound allocation request specifying an SNA service TP name. See "Unsupported TP Names" on page 230 for more information on this restriction.

A CPI Communications program that is to be started locally, by an operator for example, does not require a TP definition if the program is other than a REXX program. (See "Starting a REXX Program" on page 241 for an explanation of why REXX programs require a TP definition.)

If the program issues the Initialize_Conversation (CMINIT) call to start the TP instance, Networking Services/2 starts the TP instance on the default local LU configured for the node. If the operator (or program) wants the program to be started on a different LU, it must set the alias name for the desired local LU in the OS/2 environment variable named `APPCLLU`. Networking Services/2 then starts the program on that LU.

If the program issues the Accept_Conversation (CMACCP) call, Networking Services/2 starts the TP instance on the LU that receives the inbound allocation request.

Using Defaults for TP Definitions

Networking Services/2 provides defaults for attach manager-started programs. The defaults eliminate the need to explicitly configure TP definitions to Networking Services/2 for these programs. To use these defaults for a CPI Communications program:

- The TP name must be the same as the name of the file; Networking Services/2 converts the TP name from the inbound allocation request to ASCII and uses the ASCII name directly to start the program.
- The program must be in the OS/2 subdirectory for default TPs, or in an OS/2 subdirectory of the current path.
- The option for how default TPs are started must be `Non-queued, Attach Manager started`; the other choices are for queued (single-instance) programs, which do not apply to CPI Communications programs.
- The presentation type for the program must agree with the execution environment the program requires. The options are:
  - `Presentation Manager`
  - `VIO-windowable`
  - `Full screen`
  - `Background`

- The choice of Yes or No for whether default TPs require conversation security[7] must agree with what the CPI Communications program requires.

These default TP definitions apply across all Networking Services/2 programs—APPC programs and CPI Communications programs alike. You must explicitly configure TP definitions for all CPI Communications programs having operating characteristics that differ from these default definitions.

Running a CPI Communications Program

A CPI Communications program starts and executes as a single transaction program instance on a single LU. In addition, the scope of the *conversation_ID* is limited to one transaction program instance. Contrast this with a Networking Services/2 APPC program, which can execute as multiple transaction program instances, each on different LUs (or the same LU).

When Networking Services/2 starts a CPI Communications program, it installs itself on an OS/2 exit list for the program's OS/2 process. Then, when the program ends its execution, OS/2 gives control to the Networking Services/2 exit routine. The Networking Services/2 exit routine performs cleanup processing on behalf of the program.

With one exception, as part of its cleanup processing, Networking Services/2 deallocates dangling conversations. That is, it deallocates all remaining conversations for that program, using the *deallocate_type* of CM_DEALLOCATE_ABEND.

The exception is a program written in the REXX language. Networking Services/2 may not get control when a REXX program ends. Therefore, REXX programs should deallocate all conversations before ending. Failure to do so may require operator intervention to deactivate the sessions allocated to the conversations, in order to free the session resources for subsequent use. See "Ending a REXX Program" on page 243 for information about how to end the execution of a REXX program.

It is a good practice for all CPI Communications programs to deallocate all active conversations when they are finished with them. And, of course, programs that require their conversations to be deallocated with a *deallocate_type* other than CM_DEALLOCATE_ABEND must deallocate them before ending execution.

Networking Services/2 Use of OS/2 Environment Variables

Networking Services/2 makes use of certain OS/2 environment variables when processing CPI Communications calls. The environment variables become part of the OS/2 process created when the CPI Communications program is started. These environment variables are:

APPCTPN During an Accept_Conversation call, Networking Services/2 obtains the TP name from this environment variable. It completes the call when an inbound allocation request carrying the same TP name arrives, or if one is already waiting. The operator or program must set the TP name in this environment variable for an operator-started program. Networking Services/2 sets the TP name from an inbound allocation request in this environment variable when it starts an attach manager-started program.

[7] The option of Yes for conversation security required means the inbound allocation requests that start default TPs must include LU 6.2 access security information.

During an Initialize_Conversation call, Networking Services/2 obtains the local TP name from this environment variable when the call starts the TP instance. If the operator or program does not set this environment variable, Networking Services/2 uses `CPIC_DEFAULT_TPNAME` as a default name for the local TP instance.

`APPCLLU` During an Initialize_Conversation call that starts the TP instance, Networking Services/2 obtains the local LU name from this environment variable. It then starts the TP instance on this local LU. If this environment variable is not set when the program issues an Initialize_Conversation call that starts the TP instance, Networking Services/2 starts the TP instance on the default local LU configured for the node.

Performance Considerations in the Networking Services/2 Environment

This section provides information related to performance in the Networking Services/2 environment. Of course, programs that use these suggestions will require modification if they are to be run on an SAA system other than OS/2.

When a program issues a call that supplies a data buffer, such as Send_Data (CMSEND) and Receive (CMRCV), Networking Services/2 determines whether the data buffer is shareable across OS/2 processes. If it is, Networking Services/2 uses the buffer directly as the source or destination of data. If the buffer is not shareable, Networking Services/2 must copy the data between memory it allocates and the program's buffer. Therefore, by using shareable memory for the buffer, the program can avoid the extra copying of the data and the memory allocation that Networking Services/2 would otherwise do on each applicable call.

A program can use shareable memory for its data buffer by explicitly allocating the memory, using OS/2 calls. Depending on the application, the program may have to allocate the shareable memory only once, when it is started, and use that memory for its send and receive buffers for the remainder of its execution.

Networking Services/2 APPC Programming Reference provides detailed information on how to allocate memory for send and receive buffers, and it gives other suggestions on how to improve performance.

Programming Languages Available for CPI Communications

This section lists the SAA programming languages available with Networking Services/2 that may be used to write CPI Communications programs. This section also includes notes on the use of these languages and the pseudonym files that Networking Services/2 provides to assist the programmer.

The following SAA languages may be used to issue CPI Communications calls as well as the additional Networking Services/2 calls:

- C
- COBOL
- FORTRAN[8]
- REXX (SAA Procedures Language)

[8] FORTRAN may be used to issue all of the Networking Services/2 calls except Set_CPIC_Side_Information (XCMSSI), Delete_CPIC_Side_Information (XCMDSI), and Extract_CPIC_Side_Information (XCMESI).

The C, COBOL, and FORTRAN programming languages have the capability for a program to include header files; the REXX language does not. In the following sections, the file names are listed for the files that contain pseudonym definitions and call prototypes. By including these files, a program can use these definitions and prototypes. Listings of the pseudonym files are found in Appendix L, "Pseudonym Files" and "Pseudonym Files for Networking Services/2 Calls" on page 244, respectively, for the CPI Communications calls and the additional Networking Services/2 calls.

The C, COBOL, and FORTRAN link libraries must be defined to the system in order to link CPI Communications programs written in these languages. The link library contains the linkage edit files that Networking Services/2 provides for each of these languages. The link library is contained in the `C:\CMLIB\APPN\LIB` OS/2 subdirectory. This subdirectory path must be added to the OS/2 `SET LIB` command for your system (or environment), or it must be prefixed to the library file name in the list of libraries on the OS/2 `LINK` statement.

Additional information about the use of each of the individual SAA programming languages for CPI Communications follows.

C

The C pseudonym file provided with Networking Services/2 is:

CMC.H

This file contains the C pseudonym definitions, side information entry structure, and call prototypes for the CPI Communications calls and the additional Networking Services/2 calls. A listing of the Networking Services/2 additions to the C pseudonym file is shown in "OS/2 Additions to C Pseudonym File (CMC.H)" on page 245.

When compiling a C program using the IBM C/2 compiler, set the warning level to 3. This provides warning messages for mismatched data types and integer lengths. By including the C pseudonym file in the program, and setting the warning level to 3 for compilation, the C compiler will flag calls having incorrect variables—those with a data type (integer or character) or integer length (short or long) that does not match the corresponding call prototype statement contained in the pseudonym file.

As part of the linkage edit step for the CPI Communications program, include the `CPIC.LIB` file in the list of libraries on the OS/2 `LINK` statement.

COBOL

The COBOL pseudonym file provided with Networking Services/2 is:

CMCOBOL.CBL

This file contains the COBOL pseudonym definitions and side information entry structure for the CPI Communications calls and the additional Networking Services/2 calls. A listing of the Networking Services/2 additions to the COBOL pseudonym file is shown in "OS/2 Additions to COBOL Pseudonym File (CMCOBOL.CBL)" on page 247.

A program written to the SAA COBOL specification uses `COMP-4` integers. `COMP-4` integers in memory are in System/370 format. Therefore the program does not have to convert the value in the length (LL) field of a basic-conversation logical record to or from the System/370 format when using the value in an integer operation.

As part of the linkage edit step for the CPI Communications program written to the SAA COBOL specification, include the CPICOBOL.LIB file in the list of libraries on the OS/2 LINK statement.

A program may be written that uses COMP-5 integers; however, this provision of the IBM COBOL/2 compiler is outside the SAA COBOL specification, and such a program is not portable across all SAA systems. COMP-5 integers in memory are in byte-reversed format. This can improve performance when the program performs many integer operations. If the program uses COMP-5 integers, it should do the following:

- Specify the variables on the CPI Communications and Networking Services/2 calls in reverse order from that specified in this book.
- On the Send_Data (CMSEND) call for a basic conversation, ensure the length (LL) value is in System/370 format before issuing the call.
- Include the CPIC.LIB file in the list of libraries on the OS/2 LINK statement, and **do not** include the CPICOBOL.LIB file.

FORTRAN

The FORTRAN pseudonym file provided with Networking Services/2 is:

CMFORTRN.INC

This file contains the FORTRAN pseudonym definitions and call prototypes for the CPI Communications calls and the additional Networking Services/2 calls. A listing of the Networking Services/2 additions to the FORTRAN pseudonym file is shown in "OS/2 Additions to FORTRAN Pseudonym File (CMFORTRN.INC)" on page 248.

Three of the Networking Services/2 calls cannot be issued from a FORTRAN program. These are:

Set_CPIC_Side_Information (XCMSSI)
Extract_CPIC_Side_Information (XCMESI)
Delete_CPIC_Side_Information (XCMDSI)

As part of the linkage edit step for the CPI Communications program written to the SAA FORTRAN specification, include the CPIC.LIB file in the list of libraries on the OS/2 LINK statement.

A program may be written that uses EXTERNAL statements. To do so, the program must use the OS prefix on the EXTERNAL statements. However, use of the OS prefix is outside the SAA FORTRAN specification, and such a program is not portable across all SAA systems.

REXX (SAA Procedures Language)

The following sections are provided as an aid in writing a REXX program for the OS/2 environment.

REXX Programs

REXX programs are really command files (also called batch files) that are executed interpretively. The file extension of .CMD must be used for REXX programs.

REXX Pseudonym Definitions

The REXX pseudonym file provided with Networking Services/2 is:

```
CMREXX.CPY
```

This file contains the REXX pseudonym definitions for the CPI Communications calls and the additional Networking Services/2 calls. A listing of the Networking Services/2 additions to the REXX pseudonym file is shown in "OS/2 Additions to REXX Pseudonym File (CMREXX.CPY)" on page 249.

Unlike the other SAA languages available with Networking Services/2, the REXX language does not provide an "include" capability. Therefore, the programmer may either copy the desired pseudonym definitions directly into the program, or have REXX interpret the definitions from the pseudonym file using REXX instructions for reading and interpreting lines of a file.

Starting a REXX Program

The CPICREXX.EXE program must be executed before any REXX program that uses the CPICOMM environment can be started. The CPICREXX.EXE program can be executed by the operator, from a STARTUP.CMD file, from the CONFIG.SYS file, or by any other means the user chooses, so long as it is executed before running any REXX programs. The CPICREXX.EXE needs to be executed only once after OS/2 is started.

The CPICREXX.EXE program registers the CPICOMM environment to REXX. After the CPICOMM environment is registered, REXX programs can make CPI Communications calls.

To start a REXX program from the OS/2 command line, use the OS/2 `start` command with the `/C` command option. For example, to start a program named XCMESI.CMD from the command line, enter the following:

```
START /C XCMESI.CMD
```

This causes OS/2 to start a new OS/2 session for the CMD.EXE program, which then starts the REXX program. When the REXX program ends, OS/2 ends the session.

To start a REXX program by means of an inbound allocation request, a TP definition must be configured with CMD.EXE as the file name and the REXX program file name as the parameter string for the TP definition. Precede the file name in the parameter string with the `/C` option. For example, to start the program named XCMSSI.CMD by means of an inbound allocation request, configure the parameter string for the TP definition as:

```
/C XCMSSI.CMD
```

This causes OS/2 to start a new OS/2 session for the CMD.EXE program, which then starts the REXX program. When the REXX program ends, OS/2 ends the session.

Note: A program that is started manually from the OS/2 command line is referred to as an "operator started" program. One that is started by an inbound allocation request is referred to as an "attach manager started" program.

Making a Call

To issue a call using the REXX language, the programmer codes the following:

```
ADDRESS CPICOMM 'callname variable0 variable1 ... variableN'
```

CPICOMM is the environment name the program uses to invoke CPI Communications calls. The call name and variable names are passed as a character string to the CPICOMM environment. In Networking Services/2, the CPICOMM environment is registered to REXX by the CPICREXX.EXE program before running any REXX CPI Communications programs.

Two character-string variables used on the additional Networking Services/2 calls do not have an associated length variable. These character-string variables are:

key
sym_dest_name

These variables must be at least 8 bytes long, except for REXX programs. REXX programs may specify variables that are shorter than 8 bytes—that is, variables that have the same length as the character string they contain.

A similar exception applies to the elements of the REXX array used for the *side_info_entry* fields. REXX programs may specify array elements that are shorter than the corresponding field length—that is, elements having the same length as the character string they contain. Conversely, REXX programs may specify array elements that are longer than the corresponding *side_info_entry* field length. However, Networking Services/2 ignores any characters of an array element that are beyond the length of the corresponding field.

Checking the REXX Return Code

The CPICOMM environment uses the call name and variable names to create the actual call. In doing so, it can encounter certain error conditions prior to issuing the call. If it encounters an error, it returns to the REXX program without issuing the CPI Communications call. It sets the REXX return code variable, RC, to a non-zero value when it encounters an error; the value may be negative or positive, depending on the error. Otherwise, it issues the call, and upon completion of the call it sets the REXX return code variable to zero.

The return code values that the CPICOMM environment can return in the RC variable are shown in Table 27.

Table 27 (Page 1 of 2). Values Returned in the REXX RC Variable

RC Value	Meaning
0	The ADDRESS CPICOMM statement completed with no REXX errors.
+30	The CPICREXX.EXE program has not been executed.
−3	The CPICOMM environment does not recognize the call name specified on the ADDRESS CPICOMM statement.
−9	The CPICOMM environment requested a buffer from the CPI Communications component of Networking Services/2 in order to create the call, but insufficient buffers were available.
−10	The REXX program supplied too many variable names for the call.

Table 27 (Page 2 of 2). Values Returned in the REXX RC Variable

RC Value	Meaning
–11	The REXX program supplied too few variable names for the call.
–24	The CPICOMM environment encountered a REXX fetch failure; this usually means the REXX program supplied a name for a variable that does not exist.
–25	The CPICOMM environment encountered a REXX set failure; this means REXX memory has been exhausted.
–28	The REXX program supplied an invalid variable name.

Following the call, the REXX program should check the RC variable before it processes any values returned in the CPI Communications variables, including the CPI Communications return code. If the RC value is other than 0, the output parameters of the call are not meaningful.

Ending a REXX Program

In order for Networking Services/2 to know when a REXX program has ended, the OS/2 session in which the CMD.EXE program is running must be ended. When the session ends, control is transferred to a Networking Services/2 exit routine to perform cleanup processing. If the session is not ended, Networking Services/2 does not get control when the program is finished, and resources such as the transaction program instance and dangling conversations cannot be ended.

Following are some examples of how to cause the OS/2 session to end when the REXX program is finished:

- Use the /C option on the OS/2 start command for operator-started programs, or precede the program file name with /C on the parameter string for the TP definition.
- End the REXX program with one of the following EXIT statements:

  ```
  ADDRESS CMD 'EXIT'
  'EXIT'
  ```

  The first statement explicitly addresses the EXIT to the CMD environment, which causes the CMD.EXE session to end. The second statement is equivalent to the first, and can be used to end the OS/2 session if the current environment is the CMD environment—that is, if the program has not permanently changed the environment from the time it started.
- For operator-started programs, end the session by entering the OS/2 exit command.

Defining or Referencing a Side Information Entry

The Set_CPIC_Side_Information (XCMSSI) and Extract_CPIC_Side_Information (XCMESI) calls supply a data structure as one of the variables. REXX programs cannot create the data structure as shown in the description of these calls. In order for a REXX program to issue these calls, it must do the following:

1. Use the REXX array capability to create the structure.
2. The name for the array may be of the program's choosing, such as *sideinfo*.

3. Each element of the array must consist of the array name and the *side_info_entry* field name as the array element name. For example, the first field of a side information entry is the *sym_dest_name*, so the first element of the array would be named *sideinfo.sym_dest_name*.

4. On the REXX statement for the call, supply the array name as the *side_info_entry* variable name—*sideinfo* in this example.

See "OS/2 REXX Sample Programs" on page 257 for a listing of a REXX sample program that creates an array for the *side_info_entry* structure and issues the Set_CPIC_Side_Information and Extract_CPIC_Side_Information calls.

Pseudonym Files for Networking Services/2 Calls

This section provides a listing of the additions to the pseudonym files for the calls available on Networking Services/2.

OS/2 Additions to C Pseudonym File (CMC.H)

```
/***************************************************************************/
/*                                                                         */
/* C pseudonym values, side_info_entry_structure, and function prototypes  */
/* for OS/2 product extensions.                                            */
/*                                                                         */
/***************************************************************************/

typedef CM_INT32 XC_CONVERSATION_SECURITY_TYPE;
typedef CM_INT32 XC_TP_NAME_TYPE;

/* conversation_security_type values */
#define XC_SECURITY_NONE       (XC_CONVERSATION_SECURITY_TYPE) 0
#define XC_SECURITY_SAME       (XC_CONVERSATION_SECURITY_TYPE) 1
#define XC_SECURITY_PROGRAM    (XC_CONVERSATION_SECURITY_TYPE) 2

/* TP_name_type values */
#define XC_APPLICATION_TP        (XC_TP_NAME_TYPE) 0
#define XC_SNA_SERVICE_TP        (XC_TP_NAME_TYPE) 1

/* side info structure used by xcmssi to define side info */
typedef struct side_info_entry
   {
      unsigned char    sym_dest_name[8];    /* symbolic destination name     */
      unsigned char    partner_LU_name[17];
      unsigned char    reserved[3];         /* currently not used            */
      XC_TP_NAME_TYPE  TP_name_type;        /* set to XC_APPLICATION_TP      */
                                            /*     or XC_SNA_SERVICE_TP      */
      unsigned char    TP_name[64];
      unsigned char    mode_name[8];
      XC_CONVERSATION_SECURITY_TYPE
                       conversation_security_type;
                                            /* set to XC_SECURITY_NONE       */
                                            /*   or   XC_SECURITY_SAME       */
                                            /*   or   XC_SECURITY_PROGRAM    */
      unsigned char    security_user_ID[8];
      unsigned char    security_password[8];
   } SIDE_INFO;


/* Set_CPIC_Side_Information */
CM_ENTRY xcmssi(unsigned char CM_PTR,  /* key lock                         */
                SIDE_INFO     *,       /* side info_entry (see struct above)*/
                CM_INT32 CM_PTR,       /* side_info length                 */
                CM_INT32 CM_PTR);      /* return_code                      */

/* Extract_CPIC_Side_Information */
CM_ENTRY xcmesi(CM_INT32 CM_PTR,       /* entry_number                     */
                unsigned char CM_PTR,  /* symbolic destination name 8 chars */
                SIDE_INFO     *,       /* side_info_entry (see struct above)*/
                CM_INT32 CM_PTR,       /* side_info_length                 */
                CM_INT32 CM_PTR);      /* return_code                      */

/* Delete_CPIC_Side_Information */
CM_ENTRY xcmdsi(unsigned char CM_PTR,  /* key_lock                         */
                unsigned char CM_PTR,  /* symbolic destination name 8 chars */
                CM_INT32 CM_PTR);      /* return_code                      */
```

```
/* Extract_Conversation_Security_Type */
CM_ENTRY xcecst(unsigned char  CM_PTR, /* conversation_ID                   */
                CM_INT32 CM_PTR,       /* conversation_security_type        */
                CM_INT32 CM_PTR);      /* return_code                       */

/* Extract_Conversation_Security_User_ID */
CM_ENTRY xcecsu(unsigned char  CM_PTR, /* conversation_ID                   */
                unsigned char  CM_PTR, /* user_ID                           */
                CM_INT32 CM_PTR,       /* user_ID_length                    */
                CM_INT32 CM_PTR);      /* return_code                       */

/* Set_Conversation_Security_Password */
CM_ENTRY xcscsp(unsigned char  CM_PTR, /* conversation_ID                   */
                unsigned char  CM_PTR, /* password                          */
                CM_INT32 CM_PTR,       /* password_length                   */
                CM_INT32 CM_PTR);      /* return_code                       */

/* Set_Conversation_Security_Type */
CM_ENTRY xcscst(unsigned char  CM_PTR, /* conversation_ID                   */
                CM_INT32 CM_PTR,       /* conversation_security_type        */
                CM_INT32 CM_PTR);      /* return_code                       */

/* Set_Conversation_Security_User_ID */
CM_ENTRY xcscsu(unsigned char  CM_PTR, /* conversation_ID                   */
                unsigned char  CM_PTR, /* user_ID                           */
                CM_INT32 CM_PTR,       /* user_ID_length                    */
                CM_INT32 CM_PTR);      /* return_code                       */
```

OS/2 Additions to COBOL Pseudonym File (CMCOBOL.CBL)

```
********************************************************
* COBOL pseudonym values and side_info_entry structure *
* for OS/2 product extensions.                         *
********************************************************
*
 01  SIDE-INFO-ENTRY.
     02  SI-SYM-DEST-NAME          PIC X(8).
     02  SI-PARTNER-LU-NAME        PIC X(17).
     02  SI-RESERVED               PIC X(3).
     02  SI-TP-NAME-TYPE           PIC 9(9)  COMP-4.
        88  SI-APPLICATION-TP               VALUE 0.
        88  SI-SNA-SERVICE-TP               VALUE 1.
     02  SI-TP-NAME                PIC X(64).
     02  SI-MODE-NAME              PIC X(8).
     02  SI-SECURITY-TYPE          PIC 9(9)  COMP-4.
        88  SI-SECURITY-NONE                VALUE 0.
        88  SI-SECURITY-SAME                VALUE 1.
        88  SI-SECURITY-PROGRAM             VALUE 2.
     02  SI-USER-ID                PIC X(8).
     02  SI-PASSWORD               PIC X(8).
*
 01  SIDE-INFO-LEN                 PIC  9(9) COMP-4.
*
```

OS/2 Additions to FORTRAN Pseudonym File (CMFORTRN.INC)

```
C*****************************************************************************
C*                                                                           *
C* FORTRAN pseudonym values and prototypes for OS/2 product extensions.      *
C*                                                                           *
C*****************************************************************************

C*** conversation_security_type **********************************************
C
        INTEGER XC_SECURITY_NONE               /0/
        INTEGER XC_SECURITY_SAME               /1/
        INTEGER XC_SECURITY_PROGRAM            /2/

        INTEGER SCNONE                         /0/
        INTEGER SCSAME                         /1/
        INTEGER SCPGM                          /2/

C*** TP_name_type ************************************************************
C
        INTEGER XC_NOT_SNA_SERVICE_TP          /0/
        INTEGER XC_SNA_SERVICE_TP              /1/

        INTEGER NSERTP                         /0/
        INTEGER SERTP                          /1/

C*** OS/2 call prototypes ****************************************************
C
     OS EXTERNAL XCECST
     OS EXTERNAL XCECSU
     OS EXTERNAL XCSCSP
     OS EXTERNAL XCSCST
     OS EXTERNAL XCSCSU

C*****************************************************************************
```

OS/2 Additions to REXX Pseudonym File (CMREXX.CPY)

```
/***************************************************************************/
/* REXX statements for assigning pseudonym values, and example statements  */
/* for setting up a side_info_entry structure, for OS/2 product extensions */
/***************************************************************************/

/***************************************************************************/
/* REXX statements for assigning pseudonym values for OS/2 product        */
/* extensions                                                              */
/***************************************************************************/
XC_SECURITY_NONE                = 0;   /* conversation_security_type  */
XC_SECURITY_SAME                = 1;
XC_SECURITY_PROGRAM             = 2;

XC_APPLICATION_TP               = 0;   /* TP_name_type                */
XC_SNA_SERVICE_TP               = 1;

/***************************************************************************/
/* Example of how to set up a side_info_entry structure                    */
/***************************************************************************/
/* sideinfo.sym_dest_name              = "SYMDEST1";                  */
/* sideinfo.partner_LU_name            = "ALIASLU2";                  */
/* sideinfo.TP_name                    = "MYTPN";                     */
/* sideinfo.mode_name                  = "        ";                  */
/* sideinfo.TP_name_type               = XC_APPLICATION_TP;           */
/* sideinfo.conversation_security_type = XC_SECURITY_PROGRAM;         */
/* sideinfo.security_user_id           = "MYUSERID";                  */
/* sideinfo.security_password          = "12345678";                  */
/***************************************************************************/
/* End of example                                                          */
/***************************************************************************/
```

Sample Program Listings

This section provides listings of some sample programs that show how Networking Services/2 calls are made using the SAA languages that Networking Services/2 supports.

The listings are provided for tutorial purposes to show how the calls are made. They are not intended to show complete applications or the most efficient way of performing a function.

OS/2 C Sample Programs

SETSIDE.C

```
/**********************************************************************/
/*   This program is a C language sample program which sets CPI-C     */
/*   side information.                                                */
/*                                                                    */
/*   Note:  Before running this program, the OS/2 EE Communications   */
/*          Manager keylock feature must either be secured with a     */
/*          service key of "SVCKEY" or not be secured at all.         */
/**********************************************************************/

#include <os2.h>

#include <string.h>
#include <stdio.h>
#include <cmc.h>

int main (void)

{
  SIDE_INFO side_info;
  CM_INT32  side_info_length;
  CM_RETURN_CODE rc;

  side_info_length = sizeof(SIDE_INFO);
  /*************************************************/
  /* Set fields in side_info structure             */
  /*************************************************/
  memset(&side_info,' ',sizeof(side_info));

  memcpy(side_info.sym_dest_name,"SYMDEST1",8);
  memcpy(side_info.partner_LU_name,"ALIASLU2",8);
  side_info.TP_name_type = XC_APPLICATION_TP;
  memcpy(side_info.TP_name,"MYTPN",5);
  memcpy(side_info.mode_name,"        ",8);

  side_info.conversation_security_type = XC_SECURITY_PROGRAM;
  memcpy(side_info.security_user_ID,"MYUSERID",8);
  memcpy(side_info.security_password,"XXXXXXXX",8);

  /***********************************************************/
  /* Call XCMSSI to define the side info                     */
  /***********************************************************/
  /* Note: The key must be 8 bytes long (blank pad if needed) */
  /***********************************************************/
  xcmssi("SVCKEY  ", &side_info, &side_info_length, &rc);
  printf("return code from xcmssi is: %d \n", rc);

  return(0);
}
```

OS/2 COBOL Sample Programs

DEFSIDE.CBL

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.        DEFSIDE.
***************************************************************
* THIS PROGRAM IS AN EXAMPLE OF THE FUNCTION AVAILABLE        *
* THROUGH THE CPI-C EXTENSIONS PROVIDED.                      *
*                                                             *
* PURPOSE: DEFINE CPI-C SIDE INFORMATION AND DISPLAY RESULT   *
*                                                             *
* INPUT:    SIDE-INFORMATION STRUCTURE.                       *
*                                                             *
* OUTPUT:   CPI-C SIDE INFORMATION TABLE IS UPDATED TO        *
*           REFLECT INPUT STRUCTURE.                          *
*                                                             *
* NOTE:     FOR THIS SAMPLE PROGRAM, THE KEY FIELD (TEST-KEY), *
*           SUPPORTING THE OS/2 EE COMMUNICATIONS MANAGER     *
*           KEYLOCK FEATURE, IS SET TO SPACES.  AS A RESULT,  *
*           THIS PROGRAM WILL RUN SUCCESSFULLY ONLY WHEN      *
*           THE KEYLOCK FEATURE IS NOT SECURED.               *
*                                                             *
***************************************************************
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. PS-2.
 OBJECT-COMPUTER. PS-2.
 SPECIAL-NAMES.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
 I-O-CONTROL.
*
 DATA DIVISION.
 FILE SECTION.
 WORKING-STORAGE SECTION.

 01  TEST-KEY                  PIC  X(1)   VALUE SPACES.
 01  TEST-ENTRY-NUMBER         PIC  9(9)   VALUE 0 COMP-4.

 01  CM-ERROR-DISPLAY-MSG      PIC  X(40)  VALUE SPACES.

**********************************************
* USE THE CPI COMMUNICATIONS PSEUDONYM FILES *
**********************************************
     COPY CMCOBOL.

 LINKAGE SECTION.

 EJECT
*
 PROCEDURE DIVISION.
*******************************************************************
**************************  START OF MAINLINE  ********************
*******************************************************************
 MAINLINE.

     PERFORM SIDE-INITIALIZE
        THRU SIDE-INITIALIZE-EXIT.
```

```
       PERFORM SIDE-DISPLAY
          THRU SIDE-DISPLAY-EXIT.
       PERFORM CLEANUP
          THRU CLEANUP-EXIT.
       STOP RUN.
  ******************************************************************
  ***************************  END OF MAINLINE  ********************
  ******************************************************************
  *
   SIDE-INITIALIZE.
       INITIALIZE SIDE-INFO-ENTRY REPLACING NUMERIC BY 0
                                       ALPHABETIC BY " ".
       MOVE "CREDRPT" TO SI-SYM-DEST-NAME.
  ******************************************************************
  * CHANGE THE SI-PARTNER-LU-NAME TO MATCH YOUR CONFIGURATION      *
  ******************************************************************
       MOVE "NET1.ENLU" TO SI-PARTNER-LU-NAME.
       SET SI-APPLICATION-TP TO TRUE.
       MOVE "CREDRPT " TO SI-TP-NAME.
       MOVE "#INTER" TO SI-MODE-NAME.
       MOVE 124 TO SIDE-INFO-LEN.
       SET SI-SECURITY-NONE TO TRUE.
       CALL "XCMSSI"  USING  TEST-KEY
                      SIDE-INFO-ENTRY
                      SIDE-INFO-LEN
                      CM-RETCODE.
  *
       IF CM-OK
         DISPLAY "SIDE-INFO CREATED"
       ELSE
          MOVE "FAILURE TO CREATE SIDE-INFO"
               TO CM-ERROR-DISPLAY-MSG
          PERFORM CLEANUP
             THRU CLEANUP-EXIT
       END-IF.
   SIDE-INITIALIZE-EXIT. EXIT.
  *****************************************************************
  * CLEAR THE SIDE-INFO CONTROL BLOCK FOR TESTING PURPOSES
  * THEN ISSUE DISPLAY REQUEST
  *****************************************************************
   SIDE-DISPLAY.
       INITIALIZE SIDE-INFO-ENTRY REPLACING NUMERIC BY 0
                                       ALPHABETIC BY " ".
       DISPLAY "EXTRACTING NEWLY DEFINED SIDE INFORMATION".
       MOVE "CREDRPT" TO SI-SYM-DEST-NAME.
       MOVE 124 TO SIDE-INFO-LEN.
       CALL "XCMESI"  USING TEST-ENTRY-NUMBER
                            SI-SYM-DEST-NAME
                            SIDE-INFO-ENTRY
                            SIDE-INFO-LEN
                            CM-RETCODE.
*
      IF CM-OK THEN
        DISPLAY "-------------------------"
        DISPLAY "SIDE INFORMATION OBTAINED"
        DISPLAY "-------------------------"
        DISPLAY "PARTNER TP NAME = " SI-TP-NAME
        DISPLAY "PARTNER LU NAME = " SI-PARTNER-LU-NAME
        DISPLAY "MODE NAME       = " SI-MODE-NAME
      ELSE
      MOVE "FAILURE DURING SIDE-INFO DISPLAY"
            TO CM-ERROR-DISPLAY-MSG
      PERFORM CLEANUP
      THRU CLEANUP-EXIT
    END-IF.
  SIDE-DISPLAY-EXIT. EXIT.
***********************************************
* DISPLAY EXECUTION COMPLETE OR ERROR MESSAGE *
* NOTE: CREDRPT WILL DEALLOCATE CONVERSATION  *
***********************************************
  CLEANUP.
     IF CM-ERROR-DISPLAY-MSG  = SPACES
         DISPLAY "PROGRAM: DEFSIDE EXECUTION COMPLETE"
     ELSE
         DISPLAY "DEFSIDE PROGRAM - ",
                 CM-ERROR-DISPLAY-MSG, " RC= ", CM-RETCODE.
     STOP RUN.
 CLEANUP-EXIT. EXIT.
*****************************************************************
```

DELSIDE.CBL

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.        DELSIDE.
****************************************************************
* THIS PROGRAM IS AN EXAMPLE OF THE FUNCTION AVAILABLE         *
* THROUGH THE CPI-C EXTENSIONS PROVIDED.                       *
*                                                              *
* PURPOSE: DELETE CPI-C SIDE INFORMATION AND DISPLAY RESULT    *
*                                                              *
* INPUT:   SIDE-INFORMATION STRUCTURE.                         *
*                                                              *
* OUTPUT:  CPI-C SIDE INFORMATION TABLE IS DELETED             *
*                                                              *
* NOTE:    FOR THIS SAMPLE PROGRAM, THE KEY FIELD (TEST-KEY), *
*          SUPPORTING THE OS/2 EE COMMUNICATIONS MANAGER       *
*          KEYLOCK FEATURE, IS SET TO SPACES.  AS A RESULT,    *
*          THIS PROGRAM WILL RUN SUCCESSFULLY ONLY WHEN        *
*          THE KEYLOCK FEATURE IS NOT SECURED.                 *
*                                                              *
****************************************************************
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. PS-2.
 OBJECT-COMPUTER. PS-2.
 SPECIAL-NAMES.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
 I-O-CONTROL.
*
 DATA DIVISION.
 FILE SECTION.
 WORKING-STORAGE SECTION.

 01  TEST-KEY                      PIC  X(1)   VALUE SPACES.

 01  CM-ERROR-DISPLAY-MSG          PIC  X(40)  VALUE SPACES.

**********************************************
* USE THE CPI COMMUNICATIONS PSEUDONYM FILES *
**********************************************
     COPY CMCOBOL.

 LINKAGE SECTION.

 EJECT
*
 PROCEDURE DIVISION.
*******************************************************************
**************************  START OF MAINLINE  *******************
*******************************************************************
 MAINLINE.

    PERFORM DELETE-SIDE-INFO
       THRU DELETE-SIDE-INFO-EXIT.
     PERFORM CLEANUP
        THRU CLEANUP-EXIT.
     STOP RUN.
```

```
***********************************************
* DELETE SIDE-INFO                            *
***********************************************
  DELETE-SIDE-INFO.
     MOVE "CREDRPT" TO SI-SYM-DEST-NAME.
     CALL "XCMDSI"  USING TEST-KEY
                    SI-SYM-DEST-NAME
                    CM-RETCODE.

    IF CM-OK THEN
      DISPLAY "SIDE INFO DELETED"
    ELSE
      MOVE "FAILURE TO DELETE SIDE-INFO"
            TO CM-ERROR-DISPLAY-MSG
      PERFORM CLEANUP
       THRU CLEANUP-EXIT
    END-IF.
  DELETE-SIDE-INFO-EXIT. EXIT.
***********************************************
* DISPLAY EXECUTION COMPLETE OR ERROR MESSAGE *
***********************************************
  CLEANUP.
    IF CM-OK THEN
      DISPLAY "PROGRAM: DELETE SIDE EXECUTION COMPLETE"
    ELSE
      DISPLAY "DELSIDE PROGRAM - ",
                CM-ERROR-DISPLAY-MSG, " RC= ", CM-RETCODE
      IF  CM-PROGRAM-PARAMETER-CHECK
        DISPLAY "----------------------------------------------"
        DISPLAY "THIS ERROR CAN RESULT FROM RUNNING DELSIDE"
        DISPLAY "WHEN SIDE INFORMATION HAS ALREADY BEEN DELETED"
        DISPLAY "----------------------------------------------"
      END-IF
   END-IF.
     STOP RUN.
 CLEANUP-EXIT. EXIT.
*****************************************************************
```

OS/2 REXX Sample Programs

XCMSSI.CMD

```
/*------------------------------------------------------*/
/* REXX sample program to set CPI-C side information    */
/*                                                      */
/* Notes: CPICREXX.EXE must be run at some point prior  */
/*        to running this program in order to register  */
/*        the CPICOMM environment to REXX.              */
/*        Also, before running this program, the        */
/*        OS/2 EE Communications Manager keylock feature */
/*        must either be secured with a service key of  */
/*        "svckey" or not be secured at all.            */
/*------------------------------------------------------*/

/*----------------------*/
/* set defined constants. */
/*----------------------*/
CM_OK = 0
XC_APPLICATION_TP = 0
XC_SECURITY_PROGRAM  = 2

say 'CPI-C set side information sample program'

/*-----------------------------------------*/
/* set up parameters and sideinfo structure. */
/*-----------------------------------------*/
sideinfo_len = 124
key = "svckey  "
sideinfo.sym_dest_name = "SYMDEST1"
sideinfo.partner_LU_name = "ALIASLU2"
sideinfo.TP_name =  "MYTPN"
sideinfo.mode_name = "        "
sideinfo.TP_name_type = XC_APPLICATION_TP
sideinfo.conversation_security_type = XC_SECURITY_PROGRAM
sideinfo.security_user_id = "myuserid"
sideinfo.security_password = "xxxxxxxxx"

/*-----------------------------------------------*/
/* issue the CPI-C call to the CPICOMM environment. */
/*-----------------------------------------------*/
address CPICOMM 'xcmssi key sideinfo sideinfo_len retc'

if rc = 0 & retc = CM_OK then
   do
   say '***'
   say 'CPI-C side information successfully set.'
   say '***'
   end
else
   do
   say 'Failure to set CPI-C side information.'
   if rc = 0 then
      say 'CPI-C return code =' retc
   else if rc = 30 then
      say 'CPICREXX has not been executed.'
   else
      say 'REXX return code =' rc
   end
'pause'
'exit'
```

XCMESI.CMD

```
/*-----------------------------------------------------*/
/* REXX sample program to extract CPI-C side information */
/*                                                     */
/* Note: CPICREXX.EXE must be run at some point prior  */
/*       to running this program in order to register  */
/*       the CPICOMM environment to REXX.              */
/*-----------------------------------------------------*/

/*-----------------------*/
/* Set defined constants. */
/*-----------------------*/
CM_OK = 0
CM_PROGRAM_PARAMETER_CHECK = 24
XC_SECURITY_PROGRAM = 2

say 'CPI-C extract side information sample program'

entry_number = 1
sideinfolen = 124
rc = 0
retc = CM_OK

do while rc = 0 & retc = CM_OK
address cpicomm 'xcmesi entry_number symdest sideinfo sideinfolen retc'
if rc = 0 & retc = CM_OK then
   do
   say '***'
   say '*** Extracted CPI-C side information for entry number:'
        entry_number
   say '***'
   say 'entry_number =' entry_number
   say 'sym_dest_name =' sideinfo.sym_dest_name
   say 'TP_name =' sideinfo.TP_name
   say 'TP_name_type =' sideinfo.TP_name_type
   say 'partner_LU_name =' sideinfo.partner_LU_name
   say 'mode_name =' sideinfo.mode_name
   say 'conversation_security_type =' sideinfo.conversation_security_type
   if sideinfo.conversation_security_type = XC_SECURITY_PROGRAM then
      say 'security_user_id =' sideinfo.security_user_id
   say
   end
else
   do
   if rc = 30 then
      say 'CPICREXX has not been executed.'
   else if rc <> 0 then
      do
      say 'Failure extracting CPI-C side information '
      say 'for entry number:' entry_number
      say 'REXX return code =' rc
      end
   else if retc <> CM_PROGRAM_PARAMETER_CHECK then
      do
      say 'Failure extracting CPI-C side information '
      say 'for entry number:' entry_number
      say 'CPI-C return code =' retc
      end
   else if entry_number = 1 then
      say 'No CPI-C side information.'
   else
      say 'End of CPI-C side information entries.'
   end
entry_number = entry_number + 1
end

/*-------------------------------------------------------------*/
/* Now extract side information by symbolic destination name */
/*-------------------------------------------------------------*/
use_symdest = 0
entry_number = use_symdest
symdest = "SYMDEST1"
address cpicomm 'xcmesi entry_number symdest sideinfo sideinfolen retc'

if rc = 0 & retc = CM_OK then
   do
   say '***'
   say '*** Extracted CPI-C side information for symbolic destination name:'
        symdest
   say '***'
   say 'sym_dest_name =' sideinfo.sym_dest_name
   say 'TP_name =' sideinfo.TP_name
   say 'TP_name_type =' sideinfo.TP_name_type
   say 'partner_LU_name =' sideinfo.partner_LU_name
   say 'mode_name =' sideinfo.mode_name
   say 'conversation_security_type =' sideinfo.conversation_security_type
   if sideinfo.conversation_security_type = XC_SECURITY_PROGRAM then
      say 'security_user_id =' sideinfo.security_user_id
   end
else if rc = 0 then
   do
   say 'Failure extracting CPI-C side information'
   say 'for symbolic destination name:' symdest
   say 'CPI-C return code =' retc
   end
else if rc = 30 then
   say 'CPICREXX has not been executed.'
else
   say 'REXX return code =' rc
'pause'
'exit'
```

Appendix I. CPI Communications on Operating System/400

This appendix contains information about the Operating System/400 (OS/400) implementation of CPI Communications. This information consists of

- OS/400 terms and concepts
- An overview of the OS/400 operating environment
- Additional OS/400-related notes about CPI Communications routines
- Programming language notes for writing CPI Communications programs in an OS/400 environment
- OS/400 commands and functions that can be used in the OS/400 environment. Note that a program using any of these OS/400 commands and functions cannot be moved to another system without being modified.

The information in this appendix should be read in conjunction with the CPI Communications information contained in the Application System/400 *APPC Programmer's Guide*.

OS/400 Terms and Concepts

This section explains some special terms and concepts that should be understood before writing applications for an OS/400 environment.

Jobs

Each piece of work run on the OS/400 system is called a **job**. Each job is a single, identifiable sequence of processing actions that represents a single use of the system. The basic types of jobs performed on the OS/400 system are interactive jobs, batch jobs, spooling jobs, autostart jobs, and prestart jobs.

On the OS/400 system, all user jobs operate in an environment called a **subsystem**.

Subsystems

A subsystem is a single, predefined operating environment through which the system coordinates work flow and resource usage. The OS/400 system can contain several independently operating subsystems. The run-time characteristics of a subsystem are defined in an object called a **subsystem description**. IBM supplies several subsystem descriptions that may be used with or without modification:

QINTER	Used for interactive jobs.
QBATCH	Used for batch jobs.
QBASE	Used for both interactive and communications batch jobs.
QCMN	Used for communications batch jobs.

A new subsystem description can also be defined using the Create Subsystem Description (CRTSBSD) command.

In a subsystem description, **work entries** are defined to identify the sources from which jobs can be started in that subsystem. The types of work entries are as follows:

Autostart job entry
Specifies a job that is automatically started when the subsystem is started.

Work station entry
Specifies one or a group of work stations from which interactive jobs can be started.

Job queue entry
Specifies one of the job queues from which the subsystem can select **batch jobs**. A batch job is a job that can run independently of a user at a work station.

Communications entry
Specifies one or a group of communications device descriptions from which communications batch jobs may be started. Communications batch jobs do not use job queues.

Prestart job entry
Identifies an application program to be started to wait for incoming conversation startup requests.

When a CPI Communications program issues an Allocate (CMALLC) call, the underlying LU 6.2 support sends a conversation startup request (the LU 6.2 Functional Management Header Type 5, or FMH5). When a conversation startup request is received by a system, it is called an incoming conversation. Before the OS/400 system will start a job to run the program specified by the incoming conversation, a subsystem must be defined with the appropriate work entries to process this incoming conversation, and the subsystem must be started. The subsystem used to process the incoming conversation must have a communications entry defined to identify the communications device and the remote location name (the *partner_LU_name*) on which incoming work can be received. The IBM-supplied subsystem descriptions, QBASE and QCMN, contain default communications entries that can be used for incoming conversations. Communications entries can be added or modified using the following commands:

- Add Communications Entry (ADDCMNE)
- Remove Communications Entry (RMVCMNE)
- Change Communications Entry (CHGCMNE)

Refer to the Application System/400 *Work Management Guide* and the Application System/400 *Communications Management Guide* for more information on subsystems and communications entries.

Prestarting Jobs for Incoming Conversations

To minimize the time required for a program to accept a conversation with its partner program, an OS/400 prestart job entry can be used. When a prestart job entry is used, the application program is started before a conversation startup request is received from the partner program. Each prestart job entry contains a program name, library name, user profile, and other attributes that the subsystem uses to create and manage a pool of prestart jobs.

The following must be done to use a prestart job entry:

1. Define a prestart job entry. A prestart job entry is defined, using the Add Prestart Job Entry (ADDPJE) command, in the subsystem that contains the communications entry.
2. Start the prestart job entry. The prestart job entry can be started at the same time the subsystem is started or the Start Prestart Jobs (STRPJ) command can be used.

Programs should be designed with the following considerations when a prestart job entry is used:

- To ensure that the initial processing is completed before the conversation startup request is received, a prestart job program should do as much work as possible (for example, opening database files) before issuing the Accept_Conversation (CMACCP) call.

  The Accept_Conversation call will not complete until a conversation startup request is received for the application program. When this request is received, the application program receives control with a *return_code* of CM_OK (assuming no authorization or other problems are encountered), and the application program can immediately begin processing.
- When a prestart job program has finished servicing an incoming conversation, the conversation enters the **Reset** state, and the program may then make itself available for another incoming conversation by issuing another Accept_Conversation call.
- Only resources that are used specifically for a conversation should be deallocated. For example, if a database file is used for most conversations, there is no need to close the file and then open it each time a conversation is deallocated and a new conversation is accepted.

Programs that are designed to use prestart job entries may not be portable to other system environments.

Refer to the Application System/400 *APPC Programmer's Guide* for more information about using CPI Communications with prestart job entries.

Scope of a Conversation_ID

The OS/400 CPI Communications support associates conversations with OS/400 jobs. Each conversation is assigned a *conversation_ID* that is unique within the job; however, the *conversation_ID* is not a system-wide unique value. Therefore, a *conversation_ID* in one job cannot be accessed by another job.

Multiple Conversation Support

Application programs running in an OS/400 job can allocate many conversations and communicate concurrently over these conversations. However, only one incoming conversation can exist for one OS/400 job at any time. In addition, only prestart jobs can accept subsequent incoming conversations after deallocating an initial incoming conversation.

For example, job A can accept, process, and deallocate a conversation. If job A is a prestart job, it can then issue another Accept_Conversation to wait for another incoming conversation to process. If job A is not a prestart job, it cannot process another incoming conversation.

Protected Conversations

OS/400 CPI Communications does not support conversations with a *sync_level* of CM_SYNC_POINT. Therefore, protected conversations are not available to OS/400 programs.

OS/400 Operating Environment

There are some special considerations that should be understood when writing applications for an OS/400 environment. These considerations are explained in this section and in the following sections.

Communications Side Information

For a program to establish a conversation with a partner program, CPI Communications requires initialization parameters, such as the name of the partner program and the name of the LU at the node of the partner program. These initialization parameters are stored in the side information. On the OS/400 system, the side information is referred to as **communications side information.**

A program must specify the communications side information name as the symbolic destination name (*sym_dest_name*) parameter on the Initialize_Conversation call to use the stored characteristics. If a program does not specify a side information name (that is, the *sym_dest_name* is eight space characters) it must issue the Set_Partner_LU_Name and Set_TP_Name calls. Also, when no side information name is specified, the *mode_name* characteristic for the conversation defaults to 8 space characters. To override this default, your program must issue the Set_Mode_Name call.

On the OS/400 system, the communications side information is an object of type *CSI (communications side information). It contains information that defines the remote system (for example, the remote location name, remote network identifier, and mode). The OS/400 communications side information object also contains additional information, namely, the device description, the local location name (local network LU name), and the authority. The remote network ID, remote location name, device description, and local location name are used by the OS/400 system to establish a connection to the remote system. Refer to "Special Configuration Parameters" on page 268 for a detailed description of how the OS/400 system uses these parameters.

The following table describes the information contained in the communications side information object and maps it to the OS/400 CPI Communications parameters.

Table 28. Description of Communications Side Information Object

OS/400 System Parameters	Description
RMTLOCNAME and RMTNETID	The remote location name (RMTLOCNAME) and remote network ID (RMTNETID) make up the name of the logical unit (LU) on the remote system. These parameters correspond to the *partner_LU_name*, which is defined by the CPI Communications architecture as a required characteristic for the side information. A fully qualified *partner_LU_name* is defined as the network ID concatenated by a period with the network LU name (that is, `network ID.network LU name`). On the OS/400 system, the remote network ID is the network ID, and the remote location name is the network LU name.
MODE	The name of the mode used to control the session. It is used to designate the properties for the session that will be allocated for the conversation, such as the class of service to be used on the conversation. This parameter corresponds to the CPI Communications *mode_name* characteristic. To use a *mode_name* of eight space characters, the special value of BLANK must be used. The OS/400 system does not support sending a mode name of 'BLANKƀƀƀ' to a remote system, where ƀ is a space character.
DEV	The name of the device description, which describes the characteristics of the logical connection between a local and remote location. This parameter is specific to the OS/400 system and further qualifies the connection defined by the remote location name and the remote network ID.
LCLLOCNAME	The name of the local location, which specifies the local network LU name in a network. The OS/400 environment supports multiple local location names. This parameter is specific to the OS/400 system and further qualifies the connection defined by the remote location name and the remote network ID.
PGMNAME	The name of the target program that is to be started. This corresponds to the *TP_name*, which is defined by the CPI Communications architecture as a required characteristic for the communications side information.
AUT	The authority given to users who do not have specific authority to the communications side information object. This parameter is specific to the OS/400 system.

Managing the Communications Side Information

To manage the side information object, the OS/400 system provides Control Language (CL) commands that can be used to create, display, print, change, delete, and work with the communications side information. Command prompting from a display is also provided to perform these tasks. The following is a list of the CL commands that can be used to manage the communications side information object:

OS/400 Command	Description
CRTCSI	Used to create the CPI Communications side information object.
CHGCSI	Used to change the CPI Communications side information object.

DSPCSI	Used to display or print the CPI Communications side information object.
DLTCSI	Used to delete the CPI Communications side information object.
WRKCSI	Provides a menu from which the user can create, change, display, delete, or print the CPI Communications side information object.

Specifying the Communications Side Information Commands

The following describes the parameters for the CRTCSI and CHGCSI commands and lists the values for each parameter. Refer to "Special Configuration Parameters" on page 268 for more information on how the OS/400 system uses the RMTNETID, RMTLOCNAME, DEV, LCLLOCNAME, and MODE parameters.

CSI

Specifies the name of the side information object. An object name must be specified.

The possible library values are:

***CURLIB**

The current library for the job is used to create the side information object. If no library is specified as the current library for the job, the QGPL library is used.

library-name

Specify the name of the library in which the side information object will be stored.

side information name:

Specify the name of the object that will contain the desired side information object. This is the name an application uses for the *sym_dest_name* on an Initialize_Conversation (CMINIT) call.

RMTLOCNAME

Specifies the remote location name associated with the symbolic destination name. This name is the logical unit on the remote system and corresponds to the second part of the CPI Communications *partner_LU_name*. This is a required parameter.

remote-location-name: Enter the name of the remote location that should be associated with the symbolic destination name.

TNSPGM

Specifies the name (up to 64 characters) of the transaction program to be started. This is a required parameter.

transaction-program-name: Specify the transaction program name. This is the same as the CPI Communications *TP_name.*

Notes:

1. If a conversation startup request is received on the OS/400 system with an unqualified program name (that is, it has no library name), the system uses the library list specified on the QUSRLIBL system value at the time the subsystem handling the program start request was started.
2. If a conversation startup request is received on the OS/400 system with a qualified program name, the program name can be in the form `program.library` or in the form `library/program`.

3. Program names and library names on the OS/400 system are limited to 10 characters each.

4. To specify SNA service transaction program names, enter the hexadecimal representation of the service transaction program name. For example, to specify a service transaction program name whose hexadecimal representation is 21F0F0F1, enter the following: `X'21F0F0F1'`.

DEV

Specifies the name of the device description used for the remote system.

The possible values are:

***LOC**: The device is determined by the system.

device-name: Specify the name of the device that is associated with the remote location.

LCLLOCNAME

Specifies the name of the local location.

The possible values are:

***LOC**: The local location name is determined by the system.

*NETATR: The local location name that is in the network attributes is used.

local-location-name: Specify the name of the local location. Specify the local location if a specific local location name is desired for the remote location.

MODE

Specifies the mode used to control the session. This name is the same as the CPI Communications *mode_name*.

The possible values are:

***NETATR**: The mode specified in the system network attributes is used.

BLANK: The mode name, consisting of 8 space characters, is used.

mode-name: Specify the mode name for the remote location.

Note: The reserved SNA mode names, SNASVCMG and CPSVCMG, are not allowed.

RMTNETID

Specifies the remote network ID used with the remote location. The CPI Communications *partner_LU_name*, which consists of the remote network identifier and the remote location, determines the logical unit in the network.

The possible values are:

***LOC**: The remote network ID for the remote location is used.

*NETATR: The remote network ID specified in the network attributes is used.

*NONE: The remote network has no name.

remote-network-ID: Specify a remote network ID.

AUT

Specifies the authority given to users who do not have specific authority to the object, who are not on the authorization list, and whose users' group has no specific authority to the object.

The possible values are:

***LIBCRTAUT**

Public authority for the object is taken from the CRTAUT parameter of the specified library. This value is determined at create time. If the CRTAUT value for the library changes after the object is created, the new value does not affect any existing objects.

***CHANGE**

Change authority allows the user to perform all operations on the object except those limited to the owner or controlled by object existence authority and object management authority. The user can change the object and perform basic functions on the object. Change authority provides object operational authority and all data authority.

***ALL**

All authority allows the user to perform all operations on the object except those limited to the owner or controlled by authorization list management authority. The user can control the existence of the object, specify the security for the object, change the object, and perform basic functions on the object. The user cannot transfer ownership of the object.

***USE**

Use authority allows the user to perform basic operations on the object, such as run a program or read a file. The user is prevented from changing the object. Use authority provides object operational authority and read authority.

***EXCLUDE**

Exclude authority prevents the user from accessing the object.

authorization-list

Specify the name of the authorization list whose authority is used for the side information.

TEXT

Specify text that briefly describes this object and its function.

The possible values are:

***BLANK**: No text is specified.

'description': Specify no more than 50 characters, enclosed in apostrophes.

Special Configuration Parameters

The Create Communications Side Information (CRTCSI) and the Change Communications Side Information (CHGCSI) commands can be used to specify parameters that determine the connection that will be used to establish a conversation with a remote system. The following list shows the parameters that are used to determine the connection to the remote system and the defaults that are used for these parameters if they were not specified on the side information commands.

Parameter	Default
RMTLOCNAME	No default; parameter is required.
LCLLOCNAME	*LOC
DEV	*LOC
RMTNETID	*LOC

A default value of *NETATR is used if the MODE parameter is not specified.

Remote location names are used to make application programs independent of communications devices by allowing a remote location name to be used to access a remote communications resource. A remote communications resource is represented on the OS/400 system as one or more device descriptions; therefore, a remote location name is a logical name that is used to select the device description or descriptions.

Because the remote location name is used to determine which device description should be used, it is necessary to specify the remote location name when creating any device description that can be used by APPC. These device descriptions must be manually created in all cases except when using Advanced Peer-to-Peer Networking (APPN) support. In general, APPN does not require that remote location names be configured because the system searches the network and finds the location at the remote system where the location name was defined as a local location name. A configuration list is used for special cases in which remote locations need to be configured for APPN. In either case, the system creates device descriptions containing the remote location name when APPN is being used.

The parameters specified in the OS/400 communications side information are used to select the device description, and thus the connection, to the remote system. However, the parameters are used differently when APPN support is also configured. The following sections describe how the device description on the OS/400 is selected based on the communications side information.

Selecting Non-APPN Device Descriptions

A device description is a non-APPN device description if *NO is specified for the APPN parameter on the Create Device Description (APPC) (CRTDEVAPPC) command.

More than one device description can contain the same remote location name and other matching parameter values such as local location name and remote network ID. Special values for these parameters are resolved as follows:

- If a device is specified as *LOC, any device name will match.
- If a local location name is specified as *LOC, any local location name will match.
- If a local location name is specified as *NETATR, the value is retrieved from the network attributes.
- If a remote network ID is specified as *LOC, any remote network ID name will match.
- If a remote network ID is specified as *NETATR, the value is retrieved from the network attributes.

Two choices are available for specifying the device description. A device description can be selected by the system based on the three values (remote location name, local location name, remote network ID), or a specific device description can be requested.

A specific device description can be requested by specifying the device description name in the OS/400 communications side information. If an OS/400 special value is specified for the device description in the communications side information, the system will select a device description.

If the system is allowed to select the device description, the APPC device descriptions are searched alphabetically for all descriptions that match the remote location name, local location name, and remote network ID parameters. Of the devices that match these parameters, the system selects the first device that is most likely to have a session (for example, a device that is already actively communicating with the remote system). An error occurs if the system does not find a device that has a status of active, varied on, varied on pending, or recovery pending. The Work with Configuration Status (WRKCFGSTS) command can be used to display the status of an APPC device.

Devices are selected in the following order:

1. A device with a status of active is selected (if available).

   Note: This is the only status that is considered when a program allocates a conversation using a *return_control* of CM_IMMEDIATE.

2. A device with a status of varied on or varied on pending is selected (if available).

3. A device with a status of recovery pending is selected (if available).

If no device is available that satisfies any of these status conditions, the following will occur:

- The first device that matches the location parameters is selected.
- An error *return_code* is returned to the application program.
- An error message indicating possible actions to correct the status of the chosen device is sent to the job log.

Note: Single-session devices that are in use are not chosen if devices with unused sessions are available.

If a specific device description is requested, that description will be selected if all of the following conditions are true:

- The device description contains the specified remote location name.
- The requested values for local location name and remote network ID match the corresponding parameter values in the device.
- The device is not varied off.

Requesting a specific device description is recommended in the following cases:

- When device descriptions containing the same remote location name, local location name, and remote network ID are attached to more than one controller description. Selecting the device description lets the user control which line and controller are used.
- When a specific logical unit on the host system is required by the OS/400 application program.

If a device description is not found, the request to establish a session fails. If the device description is selected, the mode parameter is then processed. The selected device description must have a valid mode configured. The mode must also be started and, once started, have available sessions. The request to establish a session fails unless all of these conditions are met. For special considerations concerning switched lines, see the Application System/400 *APPN Guide*. Special values for the mode are resolved as follows:

- *NETATR: Value for the mode is retrieved from the system network attributes.
- BLANK: Mode is represented in the network as eight space characters.

Selecting APPN Device Descriptions

A device description is an APPN device description if *YES is specified for the APPN parameter on the Create Device Description (APPC) (CRTDEVAPPC) command or if APPN creates the device description.

The OS/400 system always selects the device description when APPN is used. This selection is accomplished using the remote location name, local location name, and remote network ID parameters that were specified in the communications side information. The local location name and remote network ID parameters are first processed as follows:

- If the local location name (LCLLOCNAME parameter) is *NETATR or *LOC, then the local location name is retrieved from the network attributes.
- If the remote network ID (RMTNETID parameter) is *NETATR, *LOC, or *NONE, then the remote network ID is retrieved from the network attributes (the LCLNETID parameter).

Notes:

1. The value specified for the device description (DEV parameter) is ignored for APPN processing. If APPN must automatically create (configure) a device description, APPN will assign a name to the description.
2. Multiple devices having the same remote location name can only exist if all devices specify either *YES or *NO for the APPN parameter.

The system then attempts to find the remote location within the APPN network (specified by the remote network ID parameter) and determines a route to the remote location. The device description selected matches the remote location name, local location name, and remote network ID, and is attached to the controller description representing the first connection of the calculated route. If a device description is selected, then it will be activated (varied on) if necessary. If a device description does not exist, then one is automatically created (configured) and activated.

If an appropriate device description cannot be selected, the request to establish a session fails. If a device description is selected, the MODE parameter is then processed. The specified MODE must be configured on the system. If the mode is configured and is not already attached to the selected device description, it is attached and automatically started by the system. If the mode is already attached, it is then started if it has not already started.

If the mode is not configured, cannot be attached to the device, cannot be started, or has no sessions available, the request to establish a session fails. Special values are processed as follows:

- *NETATR: The value for the mode is retrieved from the system network attributes.
- BLANK: The mode is represented in the network as eight space characters.

Refer to the Application System/400 *APPN Guide* for more information on automatic configuration of devices and other APPN functions.

Node Services Available in the OS/400 Environment

This section discusses the node services that are available in the OS/400 environment.

Reclaim Resource Processing

A conversation is considered a resource on the OS/400 system. Therefore, the Reclaim Resource (RCLRSC) command may be used by a user or by a program to end conversations on the OS/400 system. For each conversation that is ended by the reclaim resource processing, a *return_code* of CM_DEALLOCATED_ABEND is sent to the partner program, and a message indicating that the conversation has ended is placed in the system history log. The Display Log (DSPLOG) command can be used to display or print the system history log.

The Application System/400 *CL Reference* manual contains more information on the RCLRSC command.

Ending Dangling Conversations

When a CPI Communications program ends before one of its conversations is deallocated, the conversation is considered to be a dangling conversation. The OS/400 CPI Communications support ends dangling conversations when a *job* in which the program was running ends. When the OS/400 CPI Communications support ends a dangling conversation for a job, a *return_code* of CM_DEALLOCATED_ABEND is sent to the partner program, and a message indicating that the conversation has ended is placed in the system history log. The Display Log (DSPLOG) command can be used to display or print the system history log.

OS/400 CPI Communications Support of Log_Data

A CPI Communications program may set *log_data* using the Set_Log_Data (CMSLD) call on a basic conversation. If a program has set *log_data*, the *log_data* will be sent to the partner system when one of the following occurs:

- The CPI Communications program issues a Send_Error (CMSERR) call.
- The CPI Communications program issues a Deallocate (CMDEAL) call with *deallocate_type* of CM_DEALLOCATE_ABEND.

When the OS/400 sends *log_data* to a partner system, it places a message in the system history log. This message will include the *log_data* that was sent.

When an OS/400 receives *log_data* from a partner system, it places a message in the system history log. This message will include the *log_data* that was received.

The OS/400 system history log may be viewed or printed using the Display Log (DSPLOG) command.

Return Codes

This section discusses errors that can be returned on calls to CPI Communications routines because of reasons and errors that are specific to the OS/400 system.

CM_ALLOCATE_FAILURE_RETRY

- The operator varied off the APPC device, or the APPC device was in a recovery mode and the recovery was cancelled via a command or via an answer of cancel to a recovery message on the system operator message queue.
- The OS/400 APPN support attempted to dynamically vary on the APPC device description needed for the *partner_LU_name*. A line failure or station failure may have occurred during APPN processing, and the recovery was cancelled.
- The APPC device description needed for the *partner_LU_name* is a dependent device configured to use APPN support, but the device is varied off. OS/400 APPN support does not dynamically vary on dependent devices.
- The OS/400 APPN support was not able to determine an available route to the destination specified by the *partner_LU_name*. For example, directory services encountered a link failure on a control point in a control point session.

CM_ALLOCATE_FAILURE_NO_RETRY

- The OS/400 APPN support attempted to dynamically create the APPC device description needed for the *partner_LU_name*. The attempt was not successful because of a previous user configuration error.
- The OS/400 APPN support attempted to dynamically add the mode needed for the *mode_name* to the APPC device description needed for the *partner_LU_name*. The attempt was not successful because the needed device description already has associated with it the maximum number of modes that is allowed by configuration services.
- The OS/400 APPN support attempted to start the mode needed for the *mode_name*. The attempt was not successful because of a change-number-of-sessions (CNOS) failure.
- The OS/400 APPN support encountered a route calculation error.
- The class of service (COS) specified in the mode description cannot be found. The mode description is specified by the *mode_name* conversation characteristic.
- No route exists that satisfies the COS characteristic specified by the COS parameter in the mode description. The mode description is specified by the *mode_name* conversation characteristic.
- The local network LU (local location name) that was specified in the side information cannot be found.

CM_PARAMETER_ERROR

The CM_PARAMETER_ERROR can be returned on the Allocate call (CMALLC) for the following reasons:

- The requested *partner_LU_name* does not reside on this end node, and there is no network node available to search for the *partner_LU_name*.

- The requested *partner_LU_name* cannot be located by APPN directory services.

CM_PRODUCT_SPECIFIC_ERROR

The CM_PRODUCT_SPECIFIC_ERROR is returned on the following calls for the designated reasons:

Set_Mode_Name (CMSMN)

OS/400 CPI Communications only supports the use of the following OS/400 special values when they are specified in the side information. These special values cannot be specified on a Set_Mode_Name call.

- OS/400 special value *NETATR was specified for *mode_name*.
- OS/400 special value BLANK was specified for *mode_name*.
- An unexpected OS/400 internal error occurred.

Note: The *conversation_state* does not change for this *return_code* on this call.

Set_Partner_LU_Name (CMSPLN)

OS/400 CPI Communications only supports the use of the following OS/400 special values when they are specified in the side information. These special values cannot be specified on a Set_Partner_LU_Name call.

- OS/400 special value *LOC was specified for the network ID portion of the *partner_LU_name*.
- OS/400 special value *NETATR was specified for the network ID portion of the *partner_LU_name*.
- OS/400 special value *NONE was specified for the network ID portion of the *partner_LU_name*.
- An unexpected OS/400 internal error occurred.

Note: The *conversation_state* does not change for this *return_code* on this call.

Accept_Conversation (CMACCP)

The conversation enters **Reset** state when CM_PRODUCT_SPECIFIC_ERROR is returned in a *return_code* for the following reasons:

- The program is defined in a prestart job entry (ADDPJE command) and is being ended.
- The program has already issued an ACQUIRE operation to the *REQUESTER device using an Intersystem Communications Function (ICF) file or a communications file or a mixed file.
- OS/400 CPI Communications was unable to obtain enough system storage to support the conversation.
- An unexpected OS/400 internal error occurred.

Allocate (CMALLC)

The conversation enters **Reset** state when CM_PRODUCT_SPECIFIC_ERROR is returned in the *return_code* for the this call. CM_PRODUCT_SPECIFIC_ERROR can be returned for the following reasons:

- The program is not authorized to use the APPC device description that was selected based on the *partner_LU_name* and the side information location parameters.
- The program specified a *partner_LU_name* that caused the system to select a device description that is not an APPC device. (The network LU name portion

of the *partner_LU_name* specified an OS/400 remote location name that is configured in a non-APPC device description.)

- The *partner_LU_name* requested a conversation with a network LU name that resides on the same control point, and APPN support is being used. (The OS/400 system does not support communications between two network LU names on the same control point when APPN support is used.)
- An unexpected OS/400 internal error occurred.

Initialize_Conversation (CMINIT)
The conversation enters **Reset** state when CM_PRODUCT_SPECIFIC_ERROR is returned in a *return_code* for the following reasons:

- OS/400 CPI Communications was unable to obtain enough system storage to support the conversation.
- An unexpected OS/400 internal error occurred.

Any other set or any extract call
An unexpected OS/400 internal error occurred. The state of the conversation remains unchanged.

Determining the Reason for an Error Return_Code

Each time OS/400 CPI Communications support returns an error *return_code* to a program, a message is also placed in the job log for the user's job to indicate the cause of the error. The Display Job Log (DSPJOBLOG) command can be used to display or print the job log for any OS/400 job.

Programming Language Considerations

The following SAA languages can be used on the OS/400 system to issue SAA CPI Communications calls:

- Application Generator
- SAA C/400
- SAA COBOL/400
- SAA FORTRAN/400
- Procedures Language 400/REXX
- SAA RPG/400

Specific notes for each of these languages are listed in the individual sections that follow.

Application Generator

Cross System Product (CSP) is used to implement the Application Generator Common Programming Interface.

The OS/400 system supports CSP/Application Execution but does not support CSP/Application Development. Therefore, CSP application programs cannot be developed on the OS/400 system. However, CSP programs for the OS/400 system can be written in another environment and then run on the OS/400 system. For this reason, no pseudonym file is provided for CSP on the OS/400 system.

C/400 Language

The #pragma statement is needed for each CPI Communications routine used in a program. The #pragma statements are included in the pseudonym file provided by IBM. The pseudonym file resides in library QCC, file H, member CMC.

The C/400 language is sensitive to the case of external function names. On the OS/400 system, all CPI Communications call names *must* be typed in uppercase letters. Lowercase letters in CPI Communications call names for programs that issue CPI Communications calls on another platform (for example, cmaccp) must be changed to uppercase letters.

COBOL/400 Language

A pseudonym file is provided for the CPI Communications programs written in the COBOL/400 language. The pseudonym file resides in library QLBL, file QILBINC, member CMCOBOL.

FORTRAN/400 Language

The #pragma statement is needed for each CPI Communications call used in a program. The #pragma statements are included in the pseudonym file provided by IBM. The pseudonym file resides in library QFTN, file QIFOINC, member CMFORTRN.

REXX

OS/400 REXX programs are not compiled programs, and do not exist as OS/400 objects. Therefore, to start a REXX program as a result of a program start request, the program name in the program start request must reference a CL program that contains the Start REXX Procedure (STRREXPRC) command.

The Create Command (CRTCMD) command, the Start REXX Procedure (STRREXPRC) command, and the Change Command (CHGCMD) command allow an OS/400 user to specify an initial command environment to be used when an ADDRESS statement is not coded and a command is encountered. The special value *CPICOMM can be used for the command environment keyword CMDENV on the STRREXPRC command to specify that CPI Communications is the initial command environment. The special value *CPICOMM can be used for the REXX command environment keyword REXCMDENV on the CRTCMD and CHGCMD commands to specify that CPI Communications is the initial command environment.

No pseudonym file is provided for the OS/400 REXX language.

REXX Reserved RC Variable

The OS/400 CPICOMM environment support returns the following values in the REXX RC variable:

Code	Meaning
0	The CPI Communications routine was successfully called.
-3	The routine name specified does not exist or was spelled incorrectly.
-9	Insufficient storage is available. Attempt the call again when more storage is available.

-10 Too many parameters were specified for the CPI Communications call. Refer to the detailed description for the specified call in this book to find the proper number of parameters.

-11 Not enough parameters were specified for the CPI Communications call. Refer to the detailed description for the specified call in this book to find the proper number of parameters.

-14 An internal system error occurred in the CPICOMM environment. Attempt the call again. If the condition continues, report the problem using the Analyze Problem (ANZPRB) command.

-24 An unexpected error occurred on an internal fetch variable contents call. Attempt the call again. If the condition continues, report the problem using the Analyze Problem (ANZPRB) command.

-25 An unexpected error occurred on an internal set variable contents call. Attempt the call again. If the condition continues, report the problem using the Analyze Problem (ANZPRB) command.

-28 An invalid variable name was used. Refer to the AS/400 *Procedures Language (REXX) Programmer's Guide* for information concerning valid variable names.

REXX Error and Failure Conditions

Conditions are problems or other occurrences that may arise while a REXX program is running. **Condition traps** are routines that take control when the specified conditions are met. A condition trap is enabled by using the SIGNAL ON or CALL ON instructions.

The programmer must be aware of two conditions in the OS/400 CPI Communications command environment: FAILURE and ERROR. The OS/400 CPI Communications command environment will indicate a FAILURE condition when a negative value is returned in the RC variable. If the program has enabled a condition trap for the FAILURE condition, control passes to the routine that is named to handle the FAILURE condition. If the program has not enabled a condition trap for a FAILURE condition but has enabled a condition trap for an ERROR condition, control passes to the routine that is named to handle the ERROR condition.

Note: An ERROR condition trap will never receive control from the CPI Communications command environment if a FAILURE condition trap is also enabled.

Refer to the Application System/400 *Procedures Language (REXX) Programmer's Guide* for information concerning the error and failure conditions and how to process them.

RPG/400 Language

A pseudonym file is provided for the CPI Communications programs written in the RPG/400 language. The pseudonym file resides in library QRPG, in file QIRGINC, member CMRPG.

Portability Considerations

The following are portability considerations for CPI Communications application programs:

- The CPI Communications programmer should keep in mind that OS/400 CPI Communications support does not include protected conversations. Programs that use protected conversations must be modified before they can be migrated to use OS/400 CPI Communications.

- Programs that issue Extract_Partner_LU_Name (CMEPLN) or Extract_Mode_Name (CMEMN) calls before allocating the conversation with the Allocate (CMALLC) call may need to be modified. This is because the OS/400 CPI Communications support can also return the OS/400 special values specified in the communications side information (*NETATR and *LOC, for example). Once the Allocate call is successfully issued, these special values are resolved by the CPI Communications support and are no longer returned.

  For example, if a network LU name of NEWYORK exists in a network called APPN, the OS/400 communications side information could specify a network ID of *NETATR. This means that the name of the local network will be used. Therefore, a CMEPLN call that is issued *before* the Allocate call could return *NETATR.NEWYORK. However, a CMEPLN call that is issued *after* the Allocate call could now return APPN.NEWYORK if APPN is the local network identifier in the network attributes.

- C/400 programs that use lowercase letters in CPI Communications call names on another platform (for example, cmaccp) must be changed to uppercase letters to run correctly on the OS/400.

- Programs that handle multiple incoming conversations (one at a time) may not be portable. This is because these programs are designed to issue an Accept_Conversation call multiple times to use the prestart job entry function.

Appendix J. CPI Communications on VM/ESA CMS

This appendix contains information about VM/ESA's implementation of and extensions to CPI Communications. This information consists of:

- VM/ESA terms and concepts
- An overview of the VM/ESA operating environment
- Additional VM/ESA-related notes about CPI Communications routines
- Programming language notes for writing CPI Communications programs in a VM/ESA environment
- Special routines that can be used in VM/ESA to take advantage of VM/ESA's capabilities; note, however, that a program using any of these VM/ESA extension routines cannot be moved to another system without being changed.

All of the SAA CPI Communications routines contained in the body of this book are implemented in VM/ESA.

Before using the information in this book and this appendix, the reader should be familiar with chapters 22-25 of the *VM/ESA CMS Application Development Guide*. That VM/ESA book discusses connectivity terminology, gives an overview of communications programming on VM/ESA, introduces the use of CPI Communications on VM/ESA, and describes work units and logical units of work. It also contains scenarios and example programs that help demonstrate how to use CPI Communications in VM/ESA.

VM/ESA Terms and Concepts

This section explains some special terms and concepts that should be understood before writing applications for a VM/ESA environment.

Work Units

All CPI Communications conversations are associated with CMS work units in VM/ESA. This means that any events that affect CMS work units also affect associated conversations. For example, at end-of-work unit, all CPI Communications conversations associated with that work unit are deallocated. Work units are discussed in detail in the *VM/ESA CMS Application Development Guide*.

Protected (CM_SYNC_POINT) Conversations

Setting a conversation's *sync_level* characteristic to CM_SYNC_POINT establishes a ***protected conversation***. Protected conversations enable distributed applications to take advantage of the CMS data integrity facility, Coordinated Resource Recovery (CRR). CRR coordinates commits (and rollbacks) of work among multiple ***protected resources***. These are resources whose managers adhere to a set of requirements established for participation in CRR. Changes made to two or more protected resources on the same CMS work unit are automatically committed when a commit is issued for any of the protected resources. When the protected resources reside on different LUs in a network and are accessed by partner applications, protected conversations can be used to take advantage of CRR's capabilities. Protected conversations allow CRR to guarantee that all changes for all conversation partners are either committed or rolled back as a unit. The *VM/ESA CMS Application Development Guide* contains detailed information about CMS work units and CRR. This coordination of protected resources is based on the two-phase commit protocol as specified in the Systems Network Architecture LU 6.2 synchronization point architecture. For details regarding CRR implementation of this architecture, see the *VM/ESA CMS Planning and Administration Guide*.

The following LU 6.2 sync point services are available on VM/ESA as callable services library (CSL) routines:

Sync Point Service	Commit Routine	Backout Routine
Coordinated Resource Recovery	DMSCOMM (Commit)	DMSROLLB (Rollback)
VM/ESA Resource Recovery	SRRCMIT (Commit)	SRRBACK (Backout)

Portability

Using any of the extension routines described in this appendix means that the application will require modification to run in other SAA operating environments. Applications that use only SAA interfaces and thus are portable to other operating environments are referred to here as ***SAA applications***. For specific characteristics and further information on SAA applications in VM/ESA, see “Considerations for SAA Applications in VM/ESA” on page 293.

VM/ESA Operating Environment

There are some special considerations that should be understood when writing applications for a VM/ESA environment. These are explained in this and the following sections.

Side Information

CPI Communications defines side information, which is a set of predefined values used when starting conversations. The side information is indexed by a symbolic destination name and contains target resource location information and access security information.

VM/ESA 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 29. Contents of a CMS Communications Directory File

<table>
<tr><th>Tag</th><th>What the Value on the Tag Specifies</th></tr>
<tr><td>:nick.</td><td>Eight-character symbolic destination name for the target resource.</td></tr>
<tr><td>:luname.</td><td>The <i>partner_LU_name</i> (locally known LU name) that identifies where the resource resides. This name consists of two fields of up to 8 characters each separated by one blank. The fields are an LU name qualifier (network ID) and a <i>target_LU_name.</i> The values that can be used for each depend on the connection:
<table>
<tr><th>Connection</th><th>Network ID</th><th>Target_LU_name</th></tr>
<tr><td>To private resource within the TSAF collection (group of VM systems)</td><td>*USERID</td><td>target virtual machine's user ID</td></tr>
<tr><td>To a local or global resource within the TSAF collection</td><td>*IDENT or blank</td><td>blank</td></tr>
<tr><td>Outside the TSAF collection</td><td>defined gateway name</td><td>name of partner's LU</td></tr>
</table></td></tr>
<tr><td>:tpn.</td><td>The transaction program name as it is known at the target LU. For a local or global resource, this is the resource name identified by the resource manager. For a private resource, this is the nickname specified in the private resource server virtual machine's $SERVER$ NAMES file. For a resource in the SNA network, this is the transaction program name. This resource ID cannot start with a period.</td></tr>
<tr><td>:modename.</td><td>For connections outside the TSAF collection, this field specifies the mode name for the SNA session connecting the gateway to the target LU. For connections within the TSAF collection, this field specifies a mode name of either VMINT or VMBAT, or it is omitted. Only user programs running in requester virtual machines with OPTION COMSRV specified in their CP directory entry can specify connections with a mode name of VMINT or VMBAT.</td></tr>
<tr><td>:security.</td><td>The security type of the conversation (NONE, SAME, or PGM).</td></tr>
<tr><td>:userid.[9]</td><td>The access security user ID. (This is used for security type PGM and is ignored for other security types.)</td></tr>
<tr><td>:password.[9]</td><td>The access security password. (This is used for security type PGM and is ignored for other security types.)</td></tr>
</table>

Once the communications directory file has been created, it must be put into effect by entering the SET COMDIR command. (Refer to the *VM/ESA CMS Command Reference* for details on this command.)

Note: If a program's execution must be halted using the HX command or another CMS termination routine, any communication directory files that were loaded with SET COMDIR get unloaded. If this happens, the SET COMDIR RELOAD command must be entered to get the communications directory file loaded again.

[9] 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/ESA CMS Planning and Administration Guide* explains this in detail.

Node Services in VM/ESA

Node services in VM/ESA provides functions for CPI Communications as described in the body of this book. This section describes the VM/ESA implementation of the program-termination processing function.

- Program-termination processing (abnormal)

  A program should terminate all conversations before the end of the program. However, if the program does not terminate all conversations, node services will deallocate them abnormally. These left-over conversations are referred to as dangling conversations.

  On VM/ESA, node services will deallocate all dangling conversations with APPCVM SEVER specifying a sever code of DEALLOCATE_ABEND_SVC. This applies to both mapped and basic conversations. The DEALLOCATE_ABEND_SVC sever code indicates to the partner program that node services issued the deallocate.

  The return code the CPI Communications conversation partner sees depends on the *conversation_type* characteristic. If the *conversation_type* is CM_BASIC_CONVERSATION, the partner program sees a return code of either:

  - CM_DEALLOCATED_ABEND_SVC
  - CM_DEALLOCATED_ABEND_SVC_BO.

  If the *conversation_type* is CM_MAPPED_CONVERSATION, the partner program sees a return code of either:

  - CM_RESOURCE_FAILURE_NO_RETRY
  - CM_RESOURCE_FAIL_NO_RETRY_BO.

External Interrupts

For CPI Communications to work properly in VM/ESA, external interrupts must be enabled for a user's virtual machine.

VM/ESA-Specific Notes for CPI Communications Routines

This section contains notes that should be understood before using CPI Communications routines in VM/ESA.

Allocate (CMALLC):

- If the target program is within the same TSAF collection (group of VM systems), the CMALLC call does not allocate an LU 6.2 session.

Deallocate (CMDEAL):

- If the *deallocate_type* is CM_DEALLOCATE_ABEND and Set_Log_Data (CMSLD) was previously called to specify the *log_data* characteristic for a conversation, that log data is placed in a file called CPICOMM LOGDATA A when this routine is invoked. If the partner application is also using CPI Communications on a VM system, the log data is written to the partner's CPICOMM LOGDATA A file as well. If an error is encountered while attempting to write to the CPICOMM LOGDATA A file, the log data will not be written to the file.

  If the conversation for which the *log_data* characteristic was specified is a protected conversation (*sync_level* is set to CM_SYNC_POINT), the log data may be written during sync point processing to a file called CMSCOMM LOGDATA A, rather than to the CPICOMM LOGDATA A file.

Initialize_Conversation (CMINIT):

- CPI Communications normally returns CM_PROGRAM_PARAMETER_CHECK when the *sym_dest_name* provided does not match the side information. However, CM_PROGRAM_PARAMETER_CHECK is not returned for Initialize_Conversation on VM/ESA systems when *sym_dest_name* is not recognized. If the specified *sym_dest_name* does not have a matching entry in the side information, that name is used to set the *TP_name* characteristic for the conversation. Other conversation characteristics are initialized to the values listed in the usage notes for Initialize_Conversation except in the following cases:

mode_name	set to a null string
mode_name_length	set to zero
partner_LU_name	set to a null value, which indicates *IDENT
partner_LU_name_length	set to zero
TP_name_length	set to the length of the name passed in the *sym_dest_name* parameter.

Send_Data (CMSEND):

- If Send_Data is called from a REXX program, the buffer specified on the call cannot contain more than 32767 bytes of data. A buffer exceeding this size must be partitioned and sent in units of 32767 bytes or less. This restriction applies only to REXX.

Send_Error (CMSERR):

- If Set_Log_Data (CMSLD) was previously called to specify the *log_data* characteristic for a conversation, that log data is placed in a file called CPICOMM LOGDATA A when this routine is invoked. If the partner application is also using CPI Communications on a VM system, the log data will be written to the partner's CPICOMM LOGDATA A file as well. If a problem is encountered while attempting to write to the CPICOMM LOGDATA A file, the log data will not be written to the file.

  If the conversation for which the *log_data* characteristic was specified is a protected conversation (*sync_level* is set to CM_SYNC_POINT), the log data may be written during sync point processing to a file called CMSCOMM LOGDATA A, rather than to the CPICOMM LOGDATA A file.

Set_Log_Data (CMSLD):

- The error information supplied in the *log_data* parameter is formatted by the sending LU into an Error Log general data stream (GDS) variable. 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. See the *VM/ESA CP Programming Services* book or the *VM/ESA CP Programming Services for 370* book for the format of the Log Data GDS variable in VM.

Return Codes

This section discusses errors that can be returned on calls to CPI Communication routines due to VM/ESA-specific reasons and errors that are specific to VM/ESA (product-specific errors).

VM/ESA-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/ESA can produce return codes for reasons specific to VM/ESA. The following list shows some CPI Communications return codes, along with some possible VM/ESA-specific causes:

CM_RESOURCE_FAILURE_NO_RETRY

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

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

CM_SECURITY_NOT_VALID

This code can result for the following reasons:

- The user ID trying to allocate a conversation to a private resource is not authorized on the `:list.` tag in the private server's $SERVER$ NAMES file.
- Security type on the conversation is NONE, but the remote program does not accept XC_SECURITY_NONE
- An invalid password was supplied for a conversation with security type of XC_SECURITY_PROGRAM.

CM_TP_NOT_AVAILABLE_NO_RETRY

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

- Either the local or the remote program does not have the appropriate IUCV authority in its VM directory authorization.
- 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 server virtual machine has exceeded its maximum number of connections.
- The private server virtual machine has already accepted a protected conversation with the same LUWID.

CM_TPN_NOT_RECOGNIZED

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

- The local or global server virtual machine has not issued the Identify_Resource_Manager (XCIDRM) routine.
- The private server virtual machine cannot be autologged.
- The routine specified on the `:module.` tag in the private server's $SERVER$ NAMES file is unknown.

Product-Specific Errors in VM/ESA

CPI Communications defines a return code called CM_PRODUCT_SPECIFIC_ERROR for each routine. In VM/ESA, 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. This section lists the VM/ESA messages associated with each CPI Communications routine's CM_PRODUCT_SPECIFIC_ERROR return code. Note that each message is prefixed with "CMxxxx_PRODUCT_SPECIFIC_ERROR" when it is added to the CPICOMM LOGDATA A file. "CMxxxx" identifies the routine that produced the error.

Note: If a problem is encountered while attempting to write to the CPICOMM LOGDATA A file, the product-specific error message may not be written to the file although the application has received the CM_PRODUCT_SPECIFIC_ERROR return code.

The *code* in some messages is a decimal value representing:

- The 4-digit IPRCODE returned by an APPC/VM function
- The 4-digit return code given by a CMSIUCV or HNDIUCV function
- The 5-digit CSL reason code returned by a CMSIUCV or HNDIUCV function.

IPRCODEs are documented in the "Condition Codes and Return Codes" section of the specified function of the APPCVM macro, which is described in the *VM/ESA CP Programming Services* book and the *VM/ESA CP Programming Services for 370* book. The CMSIUCV or HNDIUCV return codes, which are padded on the left if less than 4 digits, are documented in the "Return Codes" section of those macros, both of which are described in the *VM/ESA CMS Application Development Reference for Assembler*. Reason codes are documented in the *VM/ESA System Messages and Codes Reference* and the *VM/ESA System Messages and Codes for 370* book. The *hexcode* in some messages is a hexadecimal value of 4 digits returned by a function of the APPCVM macro.

Accept_Conversation (CMACCP)

- CMSIUCV ACCEPT failed with code *code*

  (*code* can be either a 4-digit return code or a 5-digit CSL reason code)
- CMSIUCV ACCEPT failed with return code *code*
- HNDIUCV SET failed with return code *code*
- Unable to get storage.

Allocate (CMALLC)

- CMSIUCV CONNECT completed with code *code*

  (*code* can be either a 4-digit return code or a 5-digit CSL reason code)
- APPCVM CONNECT completed by a sever interrupt with IPCODE *hexcode*
- The allocation cannot be to the application's own virtual machine
- Unable to set alternate user ID
- Privilege class not authorized to set alternate user ID
- Providing a *security_password* without a *security_user_id* is invalid.

Confirm (CMCFM)

- Unexpected IPRCODE *code* from APPCVM SENDCNF call.

Deallocate (CMDEAL)

- Unexpected IPRCODE *code* from APPCVM SENDCNF or SETMODFY call.

Flush (CMFLUS)

- Unexpected IPRCODE *code* from APPCVM SENDDATA call.

Initialize_Conversation (CMINIT)

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

Prepare_to_Receive (CMPTR)

- Unexpected IPRCODE *code* from APPCVM RECEIVE call.

Receive (CMRCV)

- APPCVM RECEIVE returned neither data nor status
- Unable to get storage.

Send_Data (CMSEND)

- Unexpected IPRCODE *code* from APPCVM RECEIVE call.

Send_Error (CMSERR)

- Unexpected IPRCODE *code* from APPCVM SENDERR call.

Set_Log_Data (CMSLD)

- Unable to get storage.

Set_Partner_LU_Name (CMSPLN)

- The partner LU name cannot contain a period
- A partner LU name field cannot contain more than 8 characters
- A blank in a partner LU name should only be used as a delimiter.

Protocol Errors in VM/ESA

It is possible, although unlikely, for CPI Communications to encounter a protocol error. If this condition arises in VM/ESA, a return code of either CM_RESOURCE_FAILURE_NO_RETRY or CM_RESOURCE_FAIL_NO_RETRY_BO is returned to the application, and a file called CPICOMM LOGDATA A is appended with a message line providing the error code that caused the protocol error. Note that each message is prefixed with "CMxxxx_PROTOCOL_ERROR" when it is added to the CPICOMM LOGDATA A file. "CMxxxx" identifies the routine that produced the error. If a problem is encountered while attempting to write to the CPICOMM LOGDATA A file, the protocol error message will not be written to the file.

The X'xxxx' value in the messages is one of: X'0410', X'0420', X'0510', X'0520', or X'0530'. These codes are documented in the *VM/ESA CP Programming Services* book and the *VM/ESA CP Programming Services for 370* book. All of these can occur on mapped conversations for the CPI Communications routines listed, except code X'410' does not occur on Receive (CMRCV). The X'0410', X'0420', X'0510', and X'0520' codes can occur for basic conversations for all the routines listed except Receive.

Confirm (CMCFM)

- APPCVM SENDCNF completed with IPCODE X'xxxx' and IPWHATRC X'03'.

Deallocate (CMDEAL)

- APPCVM SEVER completed with IPCODE X'xxxx' and IPWHATRC X'03'.

Prepare_To_Receive (CMPTR)

- APPCVM SENDCNF completed with IPCODE X'xxxx' and IPWHATRC X'03'.

Receive (CMRCV)

- APPCVM RECEIVE completed with IPCODE X'xxxx' and IPWHATRC X'03'.

Send_Data (CMSEND)

- APPCVM SENDDATA completed with IPCODE X'xxxx' and IPWHATRC X'03'.

Send_Error (CMSERR)

- APPCVM SENDERR completed with IPCODE X'xxxx' and IPWHATRC X'03'.

Invoking CPI Communications Routines in VM/ESA

The following SAA languages can be used in VM/ESA to call CPI Communications: Application Generator (Cross System Product implementation), C, COBOL, FORTRAN, PL/I, and REXX (SAA Procedures Language). RPG is not supported in this environment. In addition, the non-SAA languages System/370 assembler and Pascal can be used. The calling formats for assembler and Pascal are:

Assembler

```
CALL routine_name(parm1,parm2,...return_code),VL
```

Pascal

```
routine_name (parm1,parm2,...return_code);
```

If a VM/ESA 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=code
```

The possible return codes for this message, and their meanings, are as follows:

Code **Meaning**

-07 The CPI Communications routine 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. (This command is ordinarily executed as part of the standard SYSPROF EXEC.)

-08 The CPI Communications routine called has been dropped. Follow the same steps as for return code -07.

-09 Insufficient storage is available.

-10 Too many parameters were specified for the CPI Communications routine. Refer to the detailed description for the routine in this book 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 error message 1292E is received, the called routine was not invoked and the program was terminated with an abend code of X'ACB'. If the routine was called from REXX, no abend occurs and execution continues.

Programming Language Considerations

The following SAA languages can be used on VM/ESA to call SAA CPI Communications routines and the VM/ESA extension routines:

- Application Generator
- C
- COBOL
- FORTRAN
- PL/I
- REXX (SAA Procedures Language).

In addition, the following non-SAA languages can be used on VM/ESA:

- Assembler
- Pascal.

Specific notes for each of these languages are listed in the individual sections that follow. Except where otherwise specified, the COPY files referred to in the following sections are provided in the DMSGPI MACLIB. In addition, the following note applies to all the languages, except REXX:

- Prior to running a program, CMS commands must be issued in the following format to establish proper linkage with the CPI Communications routines:

```
GLOBAL TXTLIB CMSSAA
LOAD programname (AUTO
```

Application Generator

Cross System Product (CSP) is the implementing product for the Application Generator Common Programming Interface.

Please keep the following notes in mind when coding a CSP program that calls CPI Communications routines:

- By including the file called CMCSP COPY, which is provided on the system disk, a program can use the symbolic names (pseudonyms) for the various CPI Communications values.
- The CMCSP SAMPLE file provided on the system disk in VM/SP Release 6 will remain available for migration purposes only (it will not be updated in future releases).
- When defining the CALL statement from CSP to call CPI Communications routines, use the ‘NONCSP’ parameter to avoid searching the application's load file for the CPI Communications routines.

Assembler

Please keep the following notes in mind when coding an assembler program (for Assembler H) that calls CPI Communications routines:

- By including the VM/ESA-supplied file called CMHASM COPY, a program can use the symbolic names (pseudonyms) for the various CPI Communications values.

- Addresses used with CPI Communications and VM/ESA extension routines are 32-bit fields. The high-order bit, though not being used for addressing, should **not** be used for any other purpose; it should always be zero, except when designating the end of a parameter list, in which case it is set. Specifying VL on the routine call as shown on page 287 causes the high order bit to be set automatically. If the parameter list is built manually and only the address of the list is provided in the routine call, the high-order bit of the last address in the list must be set.

C

Please keep the following notes in mind when coding a C program that calls CPI Communications routines:

- By including the VM/ESA-supplied file called CMC COPY, a program can use the symbolic names (pseudonyms) for the various CPI Communications values.
- VM/ESA does not put a terminating null byte in character strings it returns. C programs must take this into consideration.
- The #pragma statement is needed for each CPI Communications routine used in a program. The #pragma statements are included in the CMC COPY pseudonym file included with VM/ESA. Note that because C is a case-sensitive language, the CPI Communications routine name must be coded exactly as specified on the #pragma statement.
- In addition, if the C Program Offering is being used:
  - To generate the program into a module, the GENMOD command must be specified as `GENMOD filename (FROM C$START)`
  - The line `LOAD C$TEXT programname DMSCSL` is needed to load your program.
- If the C Program Product is being used, the following statements in a REXX exec may be used to set up the proper environment for compiling C programs:

```
/* A REXX exec to set the loader tables and perform necessary
   global commands */
'set ldrtbls 8'
'global loadlib edclink'
'global txtlib ibmlib cmslib cmssaa'
exit
```

COBOL

Please keep the following notes in mind when coding a COBOL program that calls CPI Communications routines:

- By including the VM/ESA-supplied file called CMCOBOL COPY, a program can use the symbolic names (pseudonyms) for the various CPI Communications values.

FORTRAN

Please keep the following notes in mind when coding a FORTRAN program that calls CPI Communications routines:

- By including the VM/ESA-supplied file called CMFORTRN COPY, a program can use the symbolic names (pseudonyms) for various CPI Communications values.
- The CMFORTRN COPY file contains both long and short variable names for compatibility with previous VM releases.

Pascal

Please keep the following notes in mind when coding a Pascal program that calls CPI Communications routines:

- By creating and including a file that contains Pascal statements equating symbolic names (pseudonyms) to various CPI Communications values, a Pascal program can use the symbolic names for the various CPI Communications values. (To create this symbol file, use the VM/ESA-supplied file called CMFORTRN COPY as a model.)
- Use internal procedure statements for each CPI Communications routine being 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 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

Please keep the following notes in mind when coding a PL/I program that calls CPI Communications routines:

- By including the VM/ESA-supplied file called CMPLI COPY, programs can use the symbolic names (pseudonyms) for various CPI Communications values.

REXX (SAA Procedures Language)

Please keep the following notes in mind when coding a REXX program that calls CPI Communications routines:

- The program should include the file called CMREXX COPY, which is provided on the system disk. This file contains REXX statements that allow you to use symbolic names (pseudonyms) for various CPI Communications values.
- The CMREXX SAMPLE file provided on the system disk in VM/SP Release 6 will remain available for migration purposes only (it will not be updated in future releases).
- If Send_Data is called from a REXX program, the buffer specified on the call cannot contain more than 32767 bytes of data. A buffer exceeding this size must be partitioned and sent in units of 32767 bytes or less. This restriction only applies to REXX.

Overview of VM/ESA Extension Routines

As mentioned earlier, VM/ESA provides some routines that are extensions to SAA CPI Communications. Programs using these routines will require modification to be portable to other SAA systems. However, these routines can be used to take advantage of the VM/ESA operating system. These extension routines are briefly introduced in the sections that follow.

Security

The default security value for CPI Communications conversations is SAME. VM/ESA provides a routine called Set_Conversation_Security_Type (XCSCST) that lets a program explicitly specify the security value (NONE, SAME, or PGM) for the conversation.

In addition, VM/ESA provides routines that let programs explicitly set and extract an access security user ID (Set_Conversation_Security_User_ID (XCSCSU) and Extract_Conversation_Security_User_ID (XCECSU)) and only set an access security password (Set_Conversation_Security_Password (XCSCSP)).

Resource Manager Programs

VM/ESA provides routines that allow a resource manager application to manage one or more resources and accept more than one conversation per resource. These routines are:

- Identify_Resource_Manager (XCIDRM) lets an application define the name of a resource it wishes to manage.
- Terminate_Resource_Manager (XCTRRM) lets an application end management of a resource it had previously defined with Identify_Resource_Manager.
- Wait_on_Event (XCWOE) lets any application wait for:
  - Communication from one or more conversation partners
  - Completion of an asynchronous Shared File System (SFS) request
  - Input generated by a console operator.

The resource manager application must indicate to CPI Communications its intent to manage a resource by calling the VM extension routine, Identify_Resource_Manager. Having done this, the resource manager application can call the Wait_on_Event routine to wait for allocation requests to the resource being managed. When ending the resource manager application, the Terminate_Resource_Manager routine must be called.

The resource manager application can be a program that either has been started automatically as a result of an allocation notification request or has been started by local (operator) action. When started automatically as a result of an allocation notification request, if the resource manager application wishes to declare its intent to manage a resource, it must call Identify_Resource_Manager **before** either accepting the conversation that started it (using Accept_Conversation) or initializing any conversation characteristics (using Initialize_Conversation). Otherwise, an Identify_Resource_Manager call will not be allowed.

Global and Local

For global and local resource managers, the application must be started and the resources identified (using Identify_Resource_Manager) before another application can attempt to allocate a conversation for those resources.

Private

A private resource manager virtual machine can be autologged and the private resource manager application automatically invoked as a result of a private resource connection request. The resource name is passed to the application as a parameter. For example, a REXX private resource manager can be coded like this:

```
/* this is an example of a private resource manager application */

arg resource_id     /* private resource name passed as parameter*/

resource_manager_type = xc_private  /* want a private resource      */
service_mode = xc_multiple          /* handle more than 1 at a time*/
security_level_flag = xc_reject_security_none

address cpicomm 'XCIDRM resource_name service_mode',
                'security_level_flag return_code'

if rc = 0           /* any csl errors? */
then do
  if return_code = cm_ok
  then do           /* don't have to wait for first connection */
                    /* go accept the conversation              */
    address cpicomm 'CMACCP conversation_id return_code'
    .
    .
    .
```

Note: Following a successful Identify_Resource_Manager call, the application could have called Wait_on_Event (XCWOE) before calling Accept_Conversation; the Wait_on_Event would complete immediately with *event_type* set to XC_ALLOCATION_REQUEST.

For a discussion of VM resources and resource manager programs see the *VM/ESA Connectivity Planning, Administration, and Operation* book.

Considerations for Intermediate (Communications) Servers

An intermediate server is a program that handles communications requests to a resource manager program on behalf of a user program. For example, if program A allocates a conversation to program B and program B in turn allocates a conversation to program C on behalf of program A (A→B→C), program B is considered an intermediate server.

An intermediate server, such as program B, must allocate the conversation to the remote program, program C, with a *conversation_security_type* of XC_SECURITY_SAME. The access security user ID flowed to program C depends on whether the intermediate server is considered by VM/ESA to be an SAA application (as described in the section entitled "Considerations for SAA Applications in VM/ESA" on page 293). If the intermediate server is considered to be an SAA application, then VM/ESA will always flow the user ID of the local progam (VMUSR1) to the remote partner, program C.

SAA application:

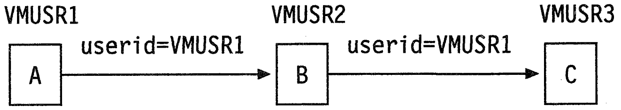


Figure 14. Access security user ID of user program flowed from VMUSR1 to VMUSR3

If the intermediate server is a not an SAA application, the access security user ID flowed to program C is the user ID of the intermediate server (VMUSR2).

Non-SAA application:

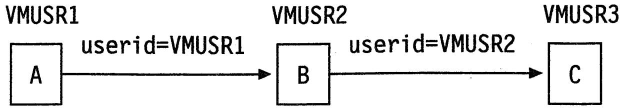


Figure 15. Access security user ID of intermediate server (VMUSR2) flowed to VMUSR3

To flow program A's user ID (VMUSR1), the intermediate server must call Set_Client_Security_User_ID.

VM/ESA Resource Recovery

CMS provides a data integrity facility called Coordinated Resource Recovery (CRR) to coordinate work among multiple protected resources. Distributed applications can take advantage of this support by using protected conversations (*sync_level* set to CM_SYNC_POINT).

To participate in CRR, a resource must be able to register an adapter with the CRR recovery server. Four routines provide functions for programming to the resource recovery adapter interface:

- Extract_Conversation_Security_User_ID (XCECSU)
- Extract_Local_Fully_Qualified_LU_Name (XCELFQ)
- Extract_Remote_Fully_Qualified_LU_Name (XCERFQ)
- Extract_TP_Name (XCETPN).

See the *VM/ESA CMS Planning and Administration Guide* for information about programming to the resource recovery adapter interface.

In addition, the Extract_Conversation_LUWID (XCECL) routine can be used to identify the most recent sync point.

CMS Work Units

Because all conversations are associated with CMS work units in VM/ESA, an extension routine, Extract_Conversation_Workunitid (XCECWU), is provided to allow applications to obtain the CMS work unit ID for a given conversation. After issuing an Allocate or Accept_Conversation call to establish a conversation, Extract_Conversation_Workunitid is used to extract the CMS work unit ID. The CMS work unit ID can be specified on such CSL routines as DMSCOMM (Commit) and DMSROLLB (Rollback).

Note that work unit manipulation is not allowed in SAA; programs that need to be portable to other SAA operating environments should use the default work unit.

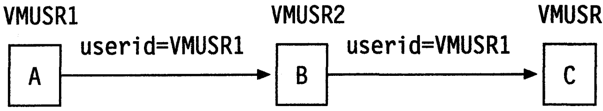
Considerations for SAA Applications in VM/ESA

As mentioned in the "VM/ESA Terms and Concepts" on page 279, applications that use only SAA interfaces are portable to other operating environments. When writing an SAA application to run in the VM/ESA operating environment, it may be helpful to keep the items discussed in this section in mind.

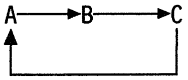
In the following discussion, the term **outbound** is used to designate a conversation allocated by an application and the term **inbound** is used to designate a conversation accepted by an application.

VM/ESA considers an application to be an SAA application if it has the following characteristics:

- It allows only one inbound conversation; there can be any number of outbound conversations.
- If there is one inbound conversation, the resource manager accepting it must not have issued an Identify_Resource_Manager (XCIDRM) call. This situation would only be possible for a private resource connection that caused CMS to automatically invoke the application. In all other cases, the application must be started locally and the application must call Identify_Resource_Manager (meaning that it is not an SAA application) and then call Wait_On_Event (XCWOE) to wait for allocation requests for the resource being managed.
- If there is one inbound conversation, any outbound conversation that uses a *conversation_security_type* of XC_SECURITY_SAME automatically propagates the user ID of the inbound conversation partner. For example, in the following case, the connection from program B to program C propagates program A's user ID:

  Note: This requires privilege class B authorization — see the description of Set_Client_Security_User_ID (XCSCUI). For a non-SAA application, Set_Client_Security_User_ID must be used by the application to propagate the user ID of the inbound conversation partner.
- For CPI Communications protected conversations (*sync_level* characteristic of CM_SYNC_POINT), the single inbound conversation is screened to prevent allocation wrap-back:

  (C wraps back to A)

  If allocation wrap-back is attempted, the connection is disallowed (deallocated by CPI Communications) and the user gets return code CM_TP_NOT_AVAILABLE_NO_RETRY. Prevention of allocation wrap-back avoids deadlock during sync point processing. This screening is done only for SAA applications.
- The application does not issue Identify_Resource_Manager, Terminate_Resource_Manager, or Set_Client_Security_User_ID. If there is one inbound conversation, then while that conversation — or any outbound conversation — is active, CPI Communications will not allow these routines to be called.

VM/ESA Extension Routines

The following table summarizes VM/ESA 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.

These routines can be used in CMS to take advantage of VM/ESA's capabilities. However, note that a program using any of these VM/ESA extension routines cannot be moved to another system without being changed.

Note: To aid in recognizing the call names, all VM/ESA extension routines begin with the prefix **XC**.

Table 30 (Page 1 of 2). Overview of VM/ESA Extension routines

Call	Pseudonym	Description	Page
XCECL	Extract_Conversation_LUWID	Lets a program extract the SNA LU 6.2 architected Logical Unit of Work ID for a given protected conversation.	297
XCECSU	Extract_Conversation_Security_User_ID	Lets a program extract the access security user ID associated with a conversation.	299
XCECWU	Extract_Conversation_Workunitid	Lets a program extract the CMS work unit ID for a given conversation.	300
XCELFQ	Extract_Local_Fully_Qualified_ LU_Name	Lets a program extract the local fully-qualified LU name for a given conversation.	301
XCERFQ	Extract_Remote_Fully_Qualified_ LU_Name	Lets a program extract the remote fully-qualified LU name for a given conversation.	302
XCETPN	Extract_TP_Name	Lets a program extract the TP name for a given conversation.	303
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 all the systems in the TSAF collection.	304
XCSCSP	Set_Conversation_Security_Password	Sets the access security password value for the conversation. The target LU uses this value and the security user ID to verify the identity of the requester.	309
XCSCST	Set_Conversation_Security_Type	Sets the security level for the conversation. The security level determines what security information is sent to the target.	311
XCSCSU	Set_Conversation_Security_User_ID	Sets the access security user ID value for the conversation. The target LU uses this value and the security password to verify the identity of the requester.	313
XCSCUI	Set_Client_Security_User_ID	Lets an intermediate server specify an alternate user ID (the user ID of a specific client application).	307
XCTRRM	Terminate_Resource_Manager	Ends ownership of a resource by a resource manager program.	315

Table 30 (Page 2 of 2). Overview of VM/ESA Extension routines

Call	Pseudonym	Description	Page
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, console input, and Shared File System (SFS) request IDs.	316

Extract_Conversation_LUWID (XCECL)

A program uses the Extract_Conversation_LUWID (XCECL) call to extract the SNA LU 6.2 architected Logical Unit of Work ID (LUWID) for a given protected conversation. The LUWID can be used to identify the most recent sync point.

This routine can be called after issuing an Allocate (CMALLC) or Accept_Conversation (CMACCP) call to establish a protected conversation.

Note: The Extract_Conversation_LUWID call is valid only for protected (*sync_level*=CM_SYNC_POINT) conversations.

Format

```
CALL XCECL(conversation_ID,
           luwid,
           luwid_length,
           return_code)
```

Parameters

conversation_ID *(input)*
specifies the conversation identifier. This variable must be an 8-byte character string.

luwid *(output)*
is a 26-byte character variable used to return the SNA LU 6.2 architected LUWID associated with the specified conversation ID when the *return_code* is CM_OK. See the usage note for a description of the LUWID.

luwid_length *(output)*
is a signed fullword variable used to return the length of the SNA LU 6.2 architected LUWID when the *return_code* is CM_OK.

return_code *(output)*
is a variable used to hold the return code passed back from the communications routine to the calling program. This return code variable must be a signed, 4-byte binary integer. The *return_code* variable can have one of the following values:

- CM_OK
  Normal completion.
- CM_PRODUCT_SPECIFIC_ERROR
  This return code can result from one of the following conditions:
  - There was a problem with the internal call to the CSL routine DMSLUWID. When this code is returned, a file named CPICOMM LOGDATA A is appended with the following line:

    ```
    XCECL_PRODUCT_SPECIFIC_ERROR:  Call to DMSLUWID failed
    ```

  - There was a problem with the internal call to the CMSIUCV QCMSWID assembler macro. When this code is returned, a file named CPICOMM LOGDATA A is appended with the following line:

    ```
    XCECL_PRODUCT_SPECIFIC_ERROR:  Call to CMSIUCV QCMSWID failed
    ```

- CM_PROGRAM_PARAMETER_CHECK
  This can result from one of the following conditions:
  - The *conversation_ID* was not found.
  - The *conversation_ID* was not for a protected (*sync_level* = CM_SYNC_POINT) conversation.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is in **Initialize** state.

State Changes

This routine does not cause a state change.

Usage Notes

1. The LUWID can be up to 26 bytes long, containing the following fields:

<table>
<tr><th>Length</th><th>Description</th></tr>
<tr><td>1</td><td>length of the fully-qualified LU name</td></tr>
<tr><td>1-17</td><td>the fully-qualified LU name. If its length is less than 17 bytes, it is left-justified and padded on the right with blanks (X'40'). It is composed of the following fields:
<table>
<tr><th>Length</th><th>Description</th></tr>
<tr><td><b>0-8</b></td><td>the network ID</td></tr>
<tr><td><b>0-1</b></td><td>a delimiter (a period)</td></tr>
<tr><td><b>1-8</b></td><td>LU name</td></tr>
</table>
If both the network ID and the LU name are present, they are separated by a period.</td></tr>
<tr><td>6</td><td>Instance number in binary</td></tr>
<tr><td>2</td><td>Sequence number in binary</td></tr>
</table>

   Refer to the *SNA Format and Protocol Reference Manual for LU Type 6.2* for more information about the LUWID.

2. This call does not change the LUWID associated with the specified conversation.

Extract_Conversation_Security_User_ID (XCECSU)

A program uses the Extract_Conversation_Security_User_ID (XCECSU) routine to extract the access security user ID associated with a conversation.

A security user ID is only returned if *conversation_security_type* is XC_SECURITY_SAME or XC_SECURITY_PROGRAM. If the *conversation_security_type* is XC_SECURITY_NONE, this variable will contain nulls (X'00'), and the length is set to zero.

The returned *security_user_ID* can be used as input to the DMSREG (Resource Adapter Registration) CSL routine, which is described in *VM/ESA CMS Administration Reference*.

Format

```
CALL XCECSU(conversation_ID,
            security_user_ID,
            security_user_ID_length,
            return_code)
```

Parameters

conversation_ID *(input)*
specifies the conversation identifier. This variable must be an 8-byte character string.

security_user_ID *(output)*
is a variable used to return the access security user ID obtained by this routine when the *return code* is CM_OK. It must be an 8-byte character string.

security_user_ID_length *(output)*
is a variable used to return the length, in bytes, of the *security_user_ID* when the *return code* is CM_OK. This variable must be declared as a 4-byte signed integer.

return_code *(output)*
is a variable used to hold the return code passed back from the communications routine to the calling program. This return code variable must be a signed, 4-byte binary integer. The *return_code* variable can have one of the following values:

- CM_OK
  Normal completion.
- CM_PROGRAM_PARAMETER_CHECK
  conversation_ID was not found.

State Changes

This routine does not cause a state change.

Usage Note

This routine does not change the conversation security user ID for the specified conversation.

Extract_Conversation_Workunitid (XCECWU)

A program uses the Extract_Conversation_Workunitid (XCECWU) call to extract the CMS work unit ID for a given conversation. This routine can be used after issuing an Allocate (CMALLC) or Accept_Conversation (CMACCP) call to establish a conversation.

The output from this routine can be used as input to specify the work unit ID on such CSL routines as DMSCOMM (Commit) and DMSROLLB (Rollback), which are described in the *VM/ESA CMS Application Development Reference*.

For information on CMS work units, refer to the publication *VM/ESA CMS Application Development Guide*.

Format

```
CALL XCECWU(conversation_ID,
            workunitid,
            return_code)
```

Parameters

conversation_ID *(input)*
specifies the conversation identifier. This variable must be an 8-byte character string.

workunitid *(output)*
is a signed 4-byte integer variable used to return the CMS work unit ID associated with the specified conversation ID when the *return_code* is CM_OK.

return_code *(output)*
is a signed 4-byte integer variable used to hold the return code passed back from the communications routine to the calling program. The *return_code* variable can have one of the following values:

- CM_OK
  Normal completion.
- CM_PRODUCT_SPECIFIC_ERROR
  There was a problem with the internal call to the CMSIUCV QCMSWID assembler macro. When this code is returned, a file named CPICOMM LOGDATA A is appended with the following line:

  ```
  XCECWU_PRODUCT_SPECIFIC_ERROR:  Call to CMSIUCV QCMSWID failed
  ```

- CM_PROGRAM_PARAMETER_CHECK
  conversation_ID was not found
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is in **Initialize** state.

State Changes

This routine does not cause a state change.

Usage Note

This call does not change the *workunitid* associated with the specified conversation.

Extract_Local_Fully_Qualified_LU_Name (XCELFQ)

A program uses the Extract_Local_Fully_Qualified_LU_Name (XCELFQ) call to extract the local fully-qualified LU name for a given conversation. This routine can be used after issuing an Allocate (CMALLC) or Accept_Conversation (CMACCP).

The output from this routine can be used as input on the DMSREG (Resource Adapter Registration) CSL routine, which is described in the *VM/ESA CMS Administration Reference*.

Format

```
CALL XCELFQ(conversation_ID,
            local_FQ_LU_name,
            local_FQ_LU_name_length,
            return_code)
```

Parameters

conversation_ID *(input)*
specifies the conversation identifier. This variable must be an 8-byte character string.

local_FQ_LU_name *(output)*
specifies the variable used to return the local fully-qualified LU name obtained by this routine when the *return_code* is CM_OK. Allow 17 bytes for this variable: 0 to 8 bytes for a network ID, 1 to 8 bytes for the LU name, and 1 byte for a delimiter if both a network ID and LU name are specified (this is a period). If the fully-qualified LU name is less than 17 bytes long, it is left-justified and padded on the right with blanks (X'40').

local_FQ_LU_name_length *(output)*
specifies the variable used to return the length of the local fully-qualified LU name obtained by this routine when the *return_code* is CM_OK. This length is zero if the partner program is on the same LU as the program issuing this routine (communication is not routed through AVS).

return_code *(output)*
specifies the return code that is passed back from the communications routine to the calling program. This return code variable must be a signed, 4-byte binary integer. The *return_code* variable can have one of the following values:

- CM_OK
  Normal completion.
- CM_PROGRAM_PARAMETER_CHECK
  conversation_ID was not found
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is in **Initialize** state.

State Changes

This routine does not cause a state change.

Usage Note

This call does not change the local fully-qualified LU name for the specified conversation.

Extract_Remote_Fully_Qualified_LU_Name (XCERFQ)

A program uses the Extract_Remote_Fully_Qualified_LU_Name (XCERFQ) call to extract the remote fully-qualified LU name for a given conversation. This routine can be used after issuing an Allocate (CMALLC) or Accept_Conversation (CMACCP).

The output from this routine can be used as input on the DMSREG (Resource Adapter Registration) CSL routine, which is described in the *VM/ESA CMS Administration Reference*.

Format

```
CALL XCERFQ(conversation_ID,
            remote_FQ_LU_name,
            remote_FQ_LU_name_length,
            return_code)
```

Parameters

conversation_ID *(input)*
specifies the conversation identifier. This variable must be an 8-byte character string.

remote_FQ_LU_name *(output)*
specifies the variable to contain the remote fully-qualified LU name obtained by this routine and returned to the local program when the *return_code* is CM_OK. Allow 17 bytes for this variable: 0 to 8 bytes for a network ID, 1 to 8 bytes for the LU name, and 1 byte for a delimiter if both a network ID and LU name are specified (this is a period). If the fully-qualified LU name is less than 17 bytes long, it is left-justified and padded on the right with blanks (X'40').

remote_FQ_LU_name_length *(output)*
specifies the variable to contain the length of the remote fully-qualified LU name obtained by this routine and returned to the local program when the *return_code* is CM_OK. This length is zero if the remote program is on the same system as the local program.

return_code *(output)*
specifies the return code that is passed back from the communications routine to the calling program. This return code variable must be a signed, 4-byte binary integer. The *return_code* variable can have one of the following values:

- CM_OK
  Normal completion.
- CM_PROGRAM_PARAMETER_CHECK
  conversation_ID was not found
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is in **Initialize** state.

State Changes

This routine does not cause a state change.

Usage Note

This call does not change the remote fully-qualified LU name for the specified conversation.

Extract_TP_Name (XCETPN)

A program uses the Extract_TP_Name (XCETPN) call to extract the *TP_name* characteristic for a given conversation. The *TP_name* characteristic can be set by using Set_TP_Name or by specifying it with a `:tpn.` tag in the communications directory file.

Format

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

Parameters

conversation_ID *(input)*
specifies the conversation identifier. This variable must be an 8-byte character string.

TP_name *(output)*
specifies the variable to contain the name of the remote program obtained by this routine and returned to the local program when the *return_code* is CM_OK. Allow 64 bytes for this variable.

TP_name_length *(output)*
specifies the variable to contain the length of the name of the remote program obtained by this routine and returned to the local program when the *return_code* is CM_OK.

return_code *(output)*
specifies the return code that is passed back from the communications routine to the calling program. This return code variable must be a signed, 4-byte binary integer. The *return_code* variable can have one of the following values:

- CM_OK
  Normal completion.
- CM_PROGRAM_PARAMETER_CHECK
  conversation_ID was not found

State Changes

This routine does not cause a state change.

Usage Notes

1. This call does not change the *TP_name* for the specified conversation.
2. When initiating a conversation, if the symbolic destination name is specified on the Initialize_Conversation (CMINIT) call, the TP name extracted by Extract_TP_Name will have been resolved using the CMS communications directory.

Identify_Resource_Manager (XCIDRM)

An application uses the Identify_Resource_Manager (XCIDRM) routine to declare to CMS the name of a resource that it 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 identified to the local system.
- For a global resource manager application, the resource name is identified to the local system and to all the CMS systems in the TSAF collection.
- For a private resource manager application, the resource name is identified only to the virtual machine where the program is running.

Format

```
CALL XCIDRM(resource_ID,
            resource_manager_type,
            service_mode,
            security_level_flag,
            return_code)
```

Parameters

resource_ID *(input)*

specifies the name of a resource managed by this resource manager application. This variable must be an 8-byte character string, padded on the right with blanks if necessary. The first character of the resource name must be alphanumeric.

The contents of the *resource_ID* parameter correspond to the transaction program name that requesting applications supply when allocating a conversation to this resource.

Other programs' allocation requests are then routed to the application that called this Identify_Resource_Manager routine.

resource_manager_type *(input)*

identifies whether the application is a private, local, or global resource manager. The *resource_manager_type* variable can have 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/ESA system, in the same TSAF collection (group of VM/ESA systems), or on another 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.

service_mode *(input)*

indicates how this resource manager application handles conversations. It can be specified with one of the following values:

- XC_SINGLE
  This resource manager program can accept only a single conversation. When it has completed processing and has deallocated the single conversation, the resource manager program should issue the Terminate_Resource_Manager (XCTRRM) call for *resource_ID* and exit.

  If the resource manager program has already accepted a conversation for the resource and another program requests that same resource, the identification of the resource as private or as local or global determines what happens to the new allocation request:

  - For a private resource, CMS queues the new allocation request. Then, when the private resource manager program ends (after the single conversation is deallocated and/or the resource manager program issues XCTRRM), the target CMS automatically restarts the private resource manager program and takes the first pending allocation request off the queue.
  - For a local or global resource, the target CMS will deallocate the allocation request. The new allocation request is not queued.

- XC_SEQUENTIAL
  This resource manager program can accept only one conversation at a time. 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 Accept_Conversation and no allocations are pending, however, a CM_PROGRAM_STATE_CHECK return code is returned.) 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 the resource manager program has already accepted a conversation for the resource and another program requests that same resource, the identification of the resource as private or as local or global determines what happens to the new allocation request:

  - For a private resource, CMS queues the new allocation request. Then after the conversation is deallocated, the resource manager program should issue Wait_on_Event or Accept_Conversation as described in the preceding paragraph.
  - For a local or global resource, the target CMS will deallocate the allocation request. The new allocation request is not queued.

- XC_MULTIPLE
  This resource manager program can accept multiple conversations.

  If the resource manager program has already accepted a conversation for the resource and another program requests the same resource, that pending allocation request is then presented the next time the resource manager issues Wait_on_Event.

security_level_flag *(input)*
indicates whether or not this resource manager will accept inbound connections that have *conversation_security_type* equal to XC_SECURITY_NONE. The *security_level_flag* variable must have 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. A program that allocates a conversation with XC_SECURITY_NONE will get a return code of CM_SECURITY_NOT_VALID upon completion of a subsequent function that allows this return code.
- XC_ACCEPT_SECURITY_NONE
  The resource manager will accept connections that have *conversation_security_type* equal to XC_SECURITY_NONE.

return_code *(output)*
specifies the return code that is passed back from the communications routine to the calling program. This return code variable must be a signed, 4-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:

  ```
  XCIDRM_PRODUCT_SPECIFIC_ERROR:  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_manager_type* contains an invalid value
  - *service_mode* contains an invalid value
  - *security_level_flag* contains an invalid value.
- CM_PROGRAM_STATE_CHECK
  This can result from the following condition:
  - This is an *SAA application*, so this routine cannot be called.
- CM_UNSUCCESSFUL
  Identify_Resource_Manager was unable to obtain ownership of the resource. The following are possible reasons:
  - The resource is already owned by another virtual machine.
  - The virtual machine in which the application is running does not have authority to connect to *IDENT.
  - The virtual machine in which the application is running does not have authority to declare the resource.

  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 a state change.

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 (XCTRRM) routine before exiting.

Set_Client_Security_User_ID (XCSCUI)

A program acting as an intermediate server uses the Set_Client_Security_User_ID (XCSCUI) routine 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 *(input)*
specifies the conversation identifier. This variable must be an 8-byte character string.

client_user_ID *(input)*
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 8-byte character string, padded on the right with blanks as necessary.

return_code *(output)*
specifies the return code that is passed back from the communications routine to the calling program. This return code variable must be a signed, 4-byte binary integer. The *return_code* variable can return one of the following values:

- CM_OK
  Normal completion.

- CM_PROGRAM_PARAMETER_CHECK
  This can result from one of the following conditions:
  - The specified *conversation_ID* was not found.
  - The *conversation_security_type* for the outgoing conversation is not equal to XC_SECURITY_SAME.
- CM_PROGRAM_STATE_CHECK
  This can result from one of the following conditions:
  - The conversation is not in **Initialize** state
  - This is an *SAA application*, so this routine cannot be called.

State Changes

This routine does not cause a state change.

Usage Notes

1. Here is a typical sequence of events that include an intermediate server calling the Set_Client_Security_User_ID:

   a. The server application issues Identify_Resource_Manager (XCIDRM) to declare the resource it is managing, and then issues Wait_on_Event (XCWOE) to wait for a request.

   b. A client application issues an Allocate (CMALLC) for a conversation to the server application.

   c. The server application accepts the conversation using the Accept_Conversation (CMACCP) routine.

   d. The server application calls the Extract_Conversation_Security_User_ID (XCECSU) routine on the conversation with the client application to get the client's access security user ID.

   e. The server calls Initialize_Conversation (CMINIT) to get a conversation ready to allocate on behalf of the client.

   f. The server application calls Set_Client_Security_User_ID using the extracted access security user ID to set the security information of the new conversation.

   g. 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.)

   h. The final target program is presented with the client program's access security user ID.

2. An intermediate server that uses XC_SECURITY_SAME and 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 cannot use Set_Client_Security_User_ID.

3. An intermediate server that uses XC_SECURITY_SAME, 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.

Set_Conversation_Security_Password (XCSCSP)

A source program or intermediate server uses the Set_Conversation_Security_Password (XCSCSP) routine to set the security password for a conversation. The password is necessary to establish a conversation with 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 after an Allocate (CMALLC).

Format

```
CALL XCSCSP(conversation_ID,
            security_password,
            security_password_length,
            return_code)
```

Parameters

conversation_ID *(input)*
specifies the conversation identifier. This variable must be an 8-byte character string.

security_password *(input)*
specifies the access security 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 upon completion of an Allocate. The *security_password* parameter must be declared as an 8-byte character variable.

security_password_length *(input)*
specifies the length of the security password. This must be a 4-byte binary integer value from 0 to 8. If 0 is specified, the password is set to null and the *security_password* parameter is ignored.

return_code *(output)*
specifies the return code that is passed back from the communications routine to the calling program. This return code variable must be a signed, 4-byte binary integer. The *return_code* variable can return one of the following values:

- CM_OK
  Normal completion.
- CM_PROGRAM_PARAMETER_CHECK
  This can result from one of the following:
  - The *conversation_ID* variable specifies an unassigned conversation ID.
  - The *security_password_length* variable specifies a value of less than 0 or greater than 8.
- CM_PROGRAM_STATE_CHECK
  This can result from one of the following:
  - The conversation is not in **Initialize** state
  - The *conversation_security_type* is not XC_SECURITY_PROGRAM.

State Changes

This routine does not cause a state change.

Usage Note

When *conversation_security_type* is XC_SECURITY_PROGRAM, both a user ID and a password are required. A program can issue this Set_Conversation_Security_Password routine and the Set_Conversation_Security_User_ID (XCSCSU) routine or have a password and user ID specified in either of the following:

- The virtual machine's communications directory file
- The virtual machine's 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. If the *security_password_length* parameter is specified as zero, however, the APPCPASS directory statement is checked. The Set_Conversation_Security_User_ID routine works the same way.

Set_Conversation_Security_Type (XCSCST)

A program uses the Set_Conversation_Security_Type (XCSCST) routine 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 after an Allocate (CMALLC).

Format

```
CALL XCSCST(conversation_ID,
            conversation_security_type,
            return_code)
```

Parameters

conversation_ID *(input)*
specifies the conversation identifier. This variable must be an 8-byte character string.

conversation_security_type *(input)*
specifies the kind of access security information to be sent to the 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. This parameter must be set 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 cannot be used for allocations outside the TSAF collection.
- XC_SECURITY_PROGRAM
  The source program must supply an access user ID and password on the allocation request.

return_code *(output)*
specifies the return code that is passed back from the communications routine to the calling program. This return code variable must be a signed, 4-byte binary integer. The *return_code* variable can return 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 a state change.

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)

A program uses the Set_Conversation_Security_User_ID (XCSCSU) routine to set the access security user ID for the conversation when the *conversation_security_type* is XC_SECURITY_PROGRAM.

Set_Conversation_Security_User_ID can only be called from **Initialize** state. It cannot be issued after an Allocate (CMALLC).

Format

```
CALL XCSCSU(conversation_ID,
            security_user_ID,
            security_user_ID_length,
            return_code)
```

Parameters

conversation_ID *(input)*
specifies the conversation ID. This variable must be an 8-byte character string.

security_user_ID *(input)*
specifies the user ID. The target LU uses this value along with the *security_password*, which can be specified on the Set_Conversation_Security_Password (XCSCSP) routine, to verify the identity of the user making the allocation request. In addition, the target LU can use the user ID for auditing or accounting purposes. This variable must be declared as an 8-byte character string.

security_user_ID_length *(input)*
specifies the length, in bytes, of the security user ID. This variable must be a 4-byte signed integer with a value from 0 to 8. If 0 is specified, the password is set to null and the *security_user_ID* parameter is ignored.

return_code *(output)*
specifies the return code that is passed back from the communications routine to the calling program. This return code variable must be a signed, 4-byte binary integer. The *return_code* variable can return one of the following values:

- CM_OK
  Normal completion.
- CM_PROGRAM_PARAMETER_CHECK
  This can result from one of the following:
  - The *conversation_ID* variable specifies an unassigned conversation ID.
  - The *security_user_ID_length* variable specifies a value less than 0 or greater than 8.
- CM_PROGRAM_STATE_CHECK
  This can result from one of the following:
  - The conversation is not in **Initialize** state
  - The *conversation_security_type* is not XC_SECURITY_PROGRAM.

State Changes

This routine does not cause a state change.

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, both a user ID and a password are required. A program can issue this Set_Conversation_Security_User_ID routine and the Set_Conversation_Security_Password (XCSCSP) routine or have a password and user ID specified in either of the following:

   - The virtual machine's communications directory file
   - The virtual machine's 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. If the *security_user_ID_length* parameter is specified as zero, however, the APPCPASS directory statement is checked. The Set_Conversation_Security_Password routine works the same way.

Terminate_Resource_Manager (XCTRRM)

A resource manager application uses the Terminate_Resource_Manager (XCTRRM) routine to end management of a resource. The resource manager automatically deallocates all conversations for the specified resource ID, and the communications 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 (group of VM systems).

Format

```
CALL XCTRRM(resource_ID,
            return_code)
```

Parameters

resource_ID *(input)*

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 (XCIDRM) routine. This is a character string variable 8 bytes in length.

return_code *(output)*

specifies the return code that is passed back from the communications routine to the calling program. This return code variable must be a signed, 4-byte binary integer. The *return_code* variable can return one of the following values:

- CM_OK
  Normal completion.
- CM_PROGRAM_PARAMETER_CHECK
  This virtual machine does not control the specified resource.
- CM_PROGRAM_STATE_CHECK
  This can result from the following condition:
  - This is an *SAA application*, so this routine cannot be called.

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

Any resource manager application that calls the Identify_Resource_Manager routine should be sure to call Terminate_Resource_Manager before exiting. Failure to call Terminate_Resource_Manager will result in the name of the resource remaining active.

Wait_on_Event (XCWOE)

An application uses the Wait_on_Event (XCWOE) routine to wait for communications from one or more partners. This routine is basically a way to handle "interrupts" from partner programs and system functions; a program can issue Wait_on_Event, then take action according to the type of interrupt it receives.

Format

```
CALL XCWOE(resource_ID,
           conversation_ID,
           event_type,
           info_input_length,
           console_input_buffer,
           return_code)
```

Parameters

resource_ID *(output)*

is a variable used to return the name of a resource managed by this resource manager application for which an event has completed. The value returned 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 8 bytes in length.

This parameter is only valid when the *event_type* is XC_ALLOCATION_REQUEST or XC_RESOURCE_REVOKED.

conversation_ID *(output)*

is a variable used to identify the conversation on which data is available to be received. It must be a character string variable 8 bytes in length.

This parameter is only valid when the *event_type* is XC_INFORMATION_INPUT. If the event is an allocation request, revoke resource request, console input, or request ID, the contents of this variable should not be examined.

event_type *(output)*

is a variable used to indicate the type of event. It must be a 4-byte signed binary integer. The *event_type* variable can return one of the following values:

- XC_ALLOCATION_REQUEST
  A program is attempting to allocate a conversation with the application that called Wait_on_Event. The application must issue an Accept_Conversation (CMACCP) to clear this event.
- XC_INFORMATION_INPUT
  The partner program is attempting to communicate information to the application that called Wait_on_Event. For instance, it might be sending data or severing its connection. The application must issue a Receive (CMRCV) to clear this event.
- XC_RESOURCE_REVOKED
  Another program has revoked the resource being managed by the application that called Wait_on_Event. In this case, the resource manager application must issue a Terminate_Resource_Manager (XCTRRM) when it completes all active conversations. This information will not be available after it has been presented to the program.

- XC_CONSOLE_INPUT
  Information is available from the console attached to this virtual machine. This information is placed in the console input buffer. It will not be available after it has been presented to the program.
- XC_REQUEST_ID
  This is the identifier for a Shared File System asynchronous event. The request ID is placed in the *info_input_length* parameter. It will not be available after it has been presented to the program. For information on SFS asynchronous events, see the *VM/ESA CMS Application Development Guide*.

info_input_length *(output)*
is a 4-byte integer variable whose contents depend on the event type as follows:

- If the *event_type* is XC_INFORMATION_INPUT, this indicates the number of data bytes that are available to be received. Note that on mapped conversations, this length will be greater than the number of bytes sent on a Send_Data (CMSEND) call. Use this value on the Receive (CMRCV). See Usage Note 2 on page 319 for details of how to use this value on a Receive.

- If the *event_type* is XC_CONSOLE_INPUT, this indicates how many bytes of data are available in the console input buffer.

- If the *event_type* is XC_REQUEST_ID, the *info_input_length* field contains the actual request ID.

If the event is an allocation request or revoke resource request, the contents of this variable should not be examined.

console_input_buffer *(output)*
is the name of a buffer for console input. On console input events, the contents of the console input buffer are stored here. This must be a character string variable with a length of 130 bytes. If more than 130 bytes were supplied, the data is truncated and the application will get only the first 130 bytes.

This parameter is only valid when *event_type* is XC_CONSOLE_INPUT.

return_code *(output)*
specifies the return code that is passed back from the communications routine to the calling program. This return code variable must be a signed, 4-byte binary integer. The *return_code* variable can return one of the following values:

- CM_OK
  Normal completion.
- CM_PRODUCT_SPECIFIC_ERROR
  This return code can result from one of the following conditions:

  - A storage failure prevented the resource manager program from being identified. A file named CPICOMM LOGDATA A is appended with the following line:

    ```
    XCWOE_PRODUCT_SPECIFIC_ERROR:  Unable to get storage
    ```

  - A storage failure prevented the resource manager program from being identified. A file named CPICOMM LOGDATA A is appended with the following line:

    ```
    XCWOE_PRODUCT_SPECIFIC_ERROR:  Unable to free storage
    ```

  - There was a problem with an internal call to the CSL routine DMSCHECK. If the problem was encountered while calling DMSCHECK, a file named CPICOMM LOGDATA A is appended with the following line:

    ```
    XCWOE_PRODUCT_SPECIFIC_ERROR:  Call to DMSCHECK failed with CSL
    return code dd
    ```

    where *dd* is a negative value. If the DMSCHECK CSL routine encountered the problem, CPICOMM LOGDATA A is appended with the following line:

    ```
    XCWOE_PRODUCT_SPECIFIC_ERROR:  Call to DMSCHECK returned reason
    code reascode
    ```

    where *reascode* is the reason code returned.

    Possible return codes and reason codes are described in the *VM/ESA CMS Application Development Reference.*

- CM_PROGRAM_STATE_CHECK
  No conversations exist and no resources were identified.

State Changes

This routine does not cause a state change.

Usage Notes

1. Events are posted in this order of priority:

   a. Allocation request
   b. Information input
   c. Resource revoke request
   d. Request ID
   e. Console input

   For example, as long as there are allocation events pending, they will be serviced first.

   - **Allocation requests**:

     After receiving the allocation event indication, the application can call the Accept_Conversation routine 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 (CMDEAL), with *deallocate_type* set to CM_DEALLOCATED_ABEND, to terminate the conversation.

   - **Information input**:

     After getting this event, the application should issue the Receive routine, using the value in the *info_input_length* parameter returned by Wait_on_Event. Note that for CM_SYNC_POINT converations, if the application initiates a backout sync point instead of issuing a Receive, the data or information associated with the event is purged unless the information is deallocation notification.

- **Resource revoke request:**

  The application will no longer receive requests for this resource. 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.

- **Request ID:**

  After receiving the request ID event indication, the application should examine the *info_input_length* field to get the request ID for a Shared File System asynchronous event. For information on SFS asynchronous events, see the *VM/ESA CMS Application Development Guide*.

- **Console Input:**

  The application determines how to interpret what is input from the console and what further action to take.

2. The *info_input_length* parameter indicates how much data is available when the *event_type* is XC_INFORMATION_INPUT or XC_CONSOLE_INPUT. When it is XC_INFORMATION_INPUT, the value of *info_input_length* should 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* to CM_RECEIVE_IMMEDIATE prior to calling the Receive.

Variables and Characteristics

The following tables are provided for the variables and characteristics used with the VM/ESA extension routines shown in this appendix:

- A chart showing the possible values for variables and characteristics associated with VM/ESA extension routines. The valid pseudonyms and corresponding integer values are provided for each variable or characteristic.
- The data definitions for types and lengths of all VM/ESA extension characteristics and variables.

Pseudonyms and Integer Values

Values for VM/ESA variables and conversation characteristics are shown as pseudonym character strings rather than integer values. For example, instead of stating that the variable *conversation_security_type* is set to an integer value of 0, this appendix shows *conversation_security_type* being set to a pseudonym value of XC_SECURITY_NONE.

Table 31 on page 321 provides a mapping from valid VM/ESA pseudonyms to integer values for each variable or characteristic.

Notes:

1. A program should set the appropriate variables equal to the appropriate values, and then use those variables as parameters on CPI Communications calls. (Some programming languages allow these integer values to be specified as literals also.)
2. Pseudonym character strings like the ones shown in the following chart should **not** be placed into variables as character strings.

For further discussion of variables, characteristics, pseudonyms, and the various naming conventions used throughout this book, please see Chapter 2, "CPI Communications Terms and Concepts" on page 11 of this book.

Table 31. VM/ESA Variables/Characteristics and their Possible Values

Variable or Characteristic Name	Values (Pseudonym Character-String)	Values (Integer)
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
	XC_REQUEST_ID	5
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

Variable Types and Lengths

Table 32 on page 322 defines the type and length of variables used specifically for VM/ESA extension routines.

Table 32. VM/ESA 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
security_password	Character string	00640	8
security_password_length	Integer	Not applicable	4
conversation_security_type	Integer	Not applicable	4
security_user_ID	Character string	00640	8
security_user_ID_length	Integer	Not applicable	4
data_length	Integer	Not applicable	4
event_type	Integer	Not applicable	4
info_input_length	Integer	Not applicable	4
luwid	Character string	00640	26
luwid_length	Integer	Not applicable	4
local_FQ_LU_name	Character string	00640	17
local_FQ_LU_name_length	Integer	Not applicable	4
remote_FQ_LU_name	Character string	00640	17
remote_FQ_LU_name_length	Integer	Not applicable	4
resource_ID	Character string	00640	8
resource_manager_type	Integer	Not applicable	4
security_level_flag	Integer	Not applicable	4
service_mode	Integer	Not applicable	4
workunitid	Integer	Not applicable	4

Appendix K. Sample Programs

This appendix contains the following sections:

- "SALESRPT (Initiator of the Conversation)" on page 324

  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 328

  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.

- "Results of Successful Program Execution" on page 333

  This section shows the output generated by the DISPLAY statements in CREDRPT and SALESRPT upon successful execution of the programs.

Both CREDRPT and SALESRPT use the various conversation characteristic values in the COBOL pseudonym file shown in Appendix L, "Pseudonym Files" on page 335. They access the pseudonym file by executing the following command:

```
COPY CMCOBOL.
```

Note: These sample programs are provided for tutorial purposes only. 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.
 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.
 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.
 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.
 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.
 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.
 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".
 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.
******************************************************************
```

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
```

Appendix L. Pseudonym Files

This appendix contains a pseudonym file for each of the SAA languages. Similar files are provided on the SAA systems as a usability aid for the CPI Communications programmer. See the appropriate product appendix for the names and locations of these files in a particular operating environment.

Application Generator Pseudonym File — as Implemented by Cross System Product (CMCSP)

The following CSP pseudonym file is provided in CSP/AD's external source format and must be imported into CSP/AD before it can be used to define CSP/AD working storage used by an application program.

```
:EZEE 330             05/09/90 11:51:43
:group     name    = C$$CCPI
           date    = '05/09/90'  time    = '11:43:50'  refine  = N
           desc    = 'Setup CPI Communication values'.
:stmts.
MOVE 2 TO CM_INITIALIZE_STATE;
MOVE 3 TO CM_SEND_STATE;
MOVE 4 TO CM_RECEIVE_STATE;
MOVE 5 TO CM_SEND_PENDING_STATE;
MOVE 6 TO CM_CONFIRM_STATE;
MOVE 7 TO CM_CONFIRM_SEND_STATE;
MOVE 8 TO CM_CONFIRM_DEALLOCATE_STATE;
MOVE 9 TO CM_DEFER_RECEIVE_STATE;
MOVE 10 TO CM_DEFER_DEALLOCATE_STATE;
MOVE 11 TO CM_SYNC_POINT_STATE;
MOVE 12 TO CM_SYNC_POINT_SEND_STATE;
MOVE 13 TO CM_SYNC_POINT_DEALLOCATE_STATE;
MOVE 0 TO CM_BASIC_CONVERSATION;
MOVE 1 TO CM_MAPPED_CONVERSATION;
MOVE 0 TO CM_OK;
MOVE 1 TO CM_ALLOCATE_FAILURE_NO_RETRY;
MOVE 2 TO CM_ALLOCATE_FAILURE_RETRY;
MOVE 3 TO CM_CONVERSATION_TYPE_MISMATCH;
MOVE 5 TO CM_PIP_NOT_SPECIFIED_CORRECTLY;
MOVE 6 TO CM_SECURITY_NOT_VALID;
MOVE 7 TO CM_SYNC_LVL_NOT_SUPPORTED_LU;
MOVE 8 TO CM_SYNC_LVL_NOT_SUPPORTED_PGM;
MOVE 9 TO CM_TPN_NOT_RECOGNIZED;
MOVE 10 TO CM_TP_NOT_AVAILABLE_NO_RETRY;
MOVE 11 TO CM_TP_NOT_AVAILABLE_RETRY;
MOVE 17 TO CM_DEALLOCATED_ABEND;
MOVE 18 TO CM_DEALLOCATED_NORMAL;
MOVE 19 TO CM_PARAMETER_ERROR;
MOVE 20 TO CM_PRODUCT_SPECIFIC_ERROR;
MOVE 21 TO CM_PROGRAM_ERROR_NO_TRUNC;
MOVE 22 TO CM_PROGRAM_ERROR_PURGING;
MOVE 23 TO CM_PROGRAM_ERROR_TRUNC;
MOVE 24 TO CM_PROGRAM_PARAMETER_CHECK;
MOVE 25 TO CM_PROGRAM_STATE_CHECK;
MOVE 26 TO CM_RESOURCE_FAILURE_NO_RETRY;
MOVE 27 TO CM_RESOURCE_FAILURE_RETRY;
MOVE 28 TO CM_UNSUCCESSFUL;
MOVE 30 TO CM_DEALLOCATED_ABEND_SVC;
MOVE 31 TO CM_DEALLOCATED_ABEND_TIMER;
MOVE 32 TO CM_SVC_ERROR_NO_TRUNC;
MOVE 33 TO CM_SVC_ERROR_PURGING;
MOVE 34 TO CM_SVC_ERROR_TRUNC;
MOVE 100 TO CM_TAKE_BACKOUT;
MOVE 130 TO CM_DEALLOCATED_ABEND_BO;
MOVE 131 TO CM_DEALLOCATED_ABEND_SVC_BO;
MOVE 132 TO CM_DEALLOCATED_ABEND_TIMER_BO;
MOVE 133 TO CM_RESOURCE_FAIL_NO_RETRY_BO;
MOVE 134 TO CM_RESOURCE_FAILURE_RETRY_BO;
MOVE 135 TO CM_DEALLOCATED_NORMAL_BO;
MOVE 0 TO CM_NO_DATA_RECEIVED;
MOVE 1 TO CM_DATA_RECEIVED;
MOVE 2 TO CM_COMPLETE_DATA_RECEIVED;
MOVE 3 TO CM_INCOMPLETE_DATA_RECEIVED;
MOVE 0 TO CM_DEALLOCATE_SYNC_LEVEL;
MOVE 1 TO CM_DEALLOCATE_FLUSH;
MOVE 2 TO CM_DEALLOCATE_CONFIRM;
MOVE 3 TO CM_DEALLOCATE_ABEND;
MOVE 0 TO CM_RECEIVE_ERROR;
MOVE 1 TO CM_SEND_ERROR;
MOVE 0 TO CM_FILL_LL;
MOVE 1 TO CM_FILL_BUFFER;
MOVE 0 TO CM_PREP_TO_RECEIVE_SYNC_LEVEL;
MOVE 1 TO CM_PREP_TO_RECEIVE_FLUSH;
MOVE 2 TO CM_PREP_TO_RECEIVE_CONFIRM;
MOVE 0 TO CM_RECEIVE_AND_WAIT;
MOVE 1 TO CM_RECEIVE_IMMEDIATE;
MOVE 0 TO CM_REQ_TO_SEND_NOT_RECEIVED;
MOVE 1 TO CM_REQ_TO_SEND_RECEIVED;
MOVE 0 TO CM_WHEN_SESSION_ALLOCATED;
MOVE 1 TO CM_IMMEDIATE;
MOVE 0 TO CM_BUFFER_DATA;
MOVE 1 TO CM_SEND_AND_FLUSH;
MOVE 2 TO CM_SEND_AND_CONFIRM;
MOVE 3 TO CM_SEND_AND_PREP_TO_RECEIVE;
MOVE 4 TO CM_SEND_AND_DEALLOCATE;
MOVE 0 TO CM_NO_STATUS_RECEIVED;
MOVE 1 TO CM_SEND_RECEIVED;
MOVE 2 TO CM_CONFIRM_RECEIVED;
MOVE 3 TO CM_CONFIRM_SEND_RECEIVED;
MOVE 4 TO CM_CONFIRM_DEALLOC_RECEIVED;
MOVE 5 TO CM_TAKE_COMMIT;
MOVE 6 TO CM_TAKE_COMMIT_SEND;
MOVE 7 TO CM_TAKE_COMMIT_DEALLOCATE;
MOVE 0 TO CM_NONE;
MOVE 1 TO CM_CONFIRM;
MOVE 2 TO CM_SYNC_POINT;
:estmts.
:egroup.
:record    name     = C$$CPIC
           date     = '05/09/90'   time = '11:45:39'
           org      = WORKSTOR
           scope    = GLOBAL
:prol.
CSP Definitions for CPI-C communications.

This working storage definition defines the variables to be
used with the communications CPI calls.  Refer to the
communications CPI call interface documentation for their
use.  Each declared constant value is defined in this
working storage record as a LEVEL 10 item binary with a
length of 4 bytes. Values used as parameters for the CPI-C
call statements must be defined in CSP as LEVEL 77 items,
and are included for the users convenience. Since CSP 3.3.0
supports the long name formats shown in the SAA CPI
Communications manual, the definitions
are shipped in this release using the same names as
```

```
defined within the CPI Communications manual. The 3.2.2
release of CSP did not support the long names, and
8 byte pseudonyms were assigned in that release.

CSP Initialization of the constant data

The statement group, name=C$$CCPI, which provides the
initialization of the constant values should be executed
before using them with the CPI Communications calls.
:eprol.
:recditem name     = CM_INITIALIZE_STATE
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_SEND_STATE
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_RECEIVE_STATE
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_SEND_PENDING_STATE
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_CONFIRM_STATE
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_CONFIRM_SEND_STATE
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_CONFIRM_DEALLOCATE_STATE
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_DEFER_RECEIVE_STATE
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_DEFER_DEALLOCATE_STATE
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_SYNC_POINT_STATE
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_SYNC_POINT_SEND_STATE
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_SYNC_POINT_DEALLOCATE_STATE
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_BASIC_CONVERSATION
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_MAPPED_CONVERSATION
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_OK
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_ALLOCATE_FAILURE_NO_RETRY
          level    = 10        occurs   = 00001
          scope    = GLOBAL
```

```
:recditem  name     = CM_ALLOCATE_FAILURE_RETRY
           level    = 10         occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_CONVERSATION_TYPE_MISMATCH
           level    = 10         occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_PIP_NOT_SPECIFIED_CORRECTLY
           level    = 10         occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_SECURITY_NOT_VALID
           level    = 10         occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_SYNC_LVL_NOT_SUPPORTED_LU
           level    = 10         occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_SYNC_LVL_NOT_SUPPORTED_PGM
           level    = 10         occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_TPN_NOT_RECOGNIZED
           level    = 10         occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_TP_NOT_AVAILABLE_NO_RETRY
           level    = 10         occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_TP_NOT_AVAILABLE_RETRY
           level    = 10         occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_DEALLOCATED_ABEND
           level    = 10         occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_DEALLOCATED_NORMAL
           level    = 10         occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_PARAMETER_ERROR
           level    = 10         occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_PRODUCT_SPECIFIC_ERROR
           level    = 10         occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_PROGRAM_ERROR_NO_TRUNC
           level    = 10         occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_PROGRAM_ERROR_PURGING
           level    = 10         occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_PROGRAM_ERROR_TRUNC
           level    = 10         occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_PROGRAM_PARAMETER_CHECK
           level    = 10         occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_PROGRAM_STATE_CHECK
           level    = 10         occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_RESOURCE_FAILURE_NO_RETRY
           level    = 10         occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_RESOURCE_FAILURE_RETRY
           level    = 10         occurs   = 00001
           scope    = GLOBAL
```

```
:recditem  name      = CM_UNSUCCESSFUL
           level     = 10         occurs    = 00001
           scope     = GLOBAL
:recditem  name      = CM_NO_DATA_RECEIVED
           level     = 10         occurs    = 00001
           scope     = GLOBAL
:recditem  name      = CM_DATA_RECEIVED
           level     = 10         occurs    = 00001
           scope     = GLOBAL
:recditem  name      = CM_COMPLETE_DATA_RECEIVED
           level     = 10         occurs    = 00001
           scope     = GLOBAL
:recditem  name      = CM_INCOMPLETE_DATA_RECEIVED
           level     = 10         occurs    = 00001
           scope     = GLOBAL
:recditem  name      = CM_DEALLOCATE_SYNC_LEVEL
           level     = 10         occurs    = 00001
           scope     = GLOBAL
:recditem  name      = CM_DEALLOCATE_FLUSH
           level     = 10         occurs    = 00001
           scope     = GLOBAL
:recditem  name      = CM_DEALLOCATE_CONFIRM
           level     = 10         occurs    = 00001
           scope     = GLOBAL
:recditem  name      = CM_DEALLOCATE_ABEND
           level     = 10         occurs    = 00001
           scope     = GLOBAL
:recditem  name      = CM_RECEIVE_ERROR
           level     = 10         occurs    = 00001
           scope     = GLOBAL
:recditem  name      = CM_SEND_ERROR
           level     = 10         occurs    = 00001
           scope     = GLOBAL
:recditem  name      = CM_FILL_LL
           level     = 10         occurs    = 00001
           scope     = GLOBAL
:recditem  name      = CM_FILL_BUFFER
           level     = 10         occurs    = 00001
           scope     = GLOBAL
:recditem  name      = CM_PREP_TO_RECEIVE_SYNC_LEVEL
           level     = 10         occurs    = 00001
           scope     = GLOBAL
:recditem  name      = CM_PREP_TO_RECEIVE_FLUSH
           level     = 10         occurs    = 00001
           scope     = GLOBAL
:recditem  name      = CM_PREP_TO_RECEIVE_CONFIRM
           level     = 10         occurs    = 00001
           scope     = GLOBAL
:recditem  name      = CM_RECEIVE_AND_WAIT
           level     = 10         occurs    = 00001
           scope     = GLOBAL
:recditem  name      = CM_RECEIVE_IMMEDIATE
           level     = 10         occurs    = 00001
           scope     = GLOBAL
:recditem  name      = CM_REQ_TO_SEND_NOT_RECEIVED
           level     = 10         occurs    = 00001
           scope     = GLOBAL
:recditem  name      = CM_REQ_TO_SEND_RECEIVED
           level     = 10         occurs    = 00001
           scope     = GLOBAL
```

```
:recditem  name     = CM_WHEN_SESSION_ALLOCATED
           level    = 10        occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_IMMEDIATE
           level    = 10        occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_BUFFER_DATA
           level    = 10        occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_SEND_AND_FLUSH
           level    = 10        occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_SEND_AND_CONFIRM
           level    = 10        occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_SEND_AND_PREP_TO_RECEIVE
           level    = 10        occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_SEND_AND_DEALLOCATE
           level    = 10        occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_NO_STATUS_RECEIVED
           level    = 10        occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_SEND_RECEIVED
           level    = 10        occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_CONFIRM_RECEIVED
           level    = 10        occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_CONFIRM_SEND_RECEIVED
           level    = 10        occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_CONFIRM_DEALLOC_RECEIVED
           level    = 10        occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_NONE
           level    = 10        occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_CONFIRM
           level    = 10        occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_SYNC_POINT
           level    = 10        occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_DEALLOCATED_ABEND_SVC
           level    = 10        occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_DEALLOCATED_ABEND_TIMER
           level    = 10        occurs   = 00001
           scope    = GLOBAL
```

```
:recditem name     = CM_SVC_ERROR_NO_TRUNC
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_SVC_ERROR_PURGING
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_SVC_ERROR_TRUNC
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_TAKE_BACKOUT
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_DEALLOCATED_ABEND_BO
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_DEALLOCATED_ABEND_SVC_BO
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_DEALLOCATED_ABEND_TIMER_BO
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_RESOURCE_FAIL_NO_RETRY_BO
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_RESOURCE_FAILURE_RETRY_BO
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_DEALLOCATED_NORMAL_BO
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_TAKE_COMMIT
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_TAKE_COMMIT_SEND
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CM_TAKE_COMMIT_DEALLOCATE
          level    = 10        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CONVERSATION_ID
          level    = 77        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CONVERSATION_STATE
          level    = 77        occurs   = 00001
          scope    = GLOBAL
:recditem name     = CONVERSATION_TYPE
          level    = 77        occurs   = 00001
          scope    = GLOBAL
:recditem name     = RETURN_CODE
          level    = 77        occurs   = 00001
          scope    = GLOBAL
:recditem name     = DATA_RECEIVED
          level    = 77        occurs   = 00001
          scope    = GLOBAL
:recditem name     = DEALLOCATE_TYPE
          level    = 77        occurs   = 00001
          scope    = GLOBAL
```

```
:recditem  name     = ERROR_DIRECTION
           level    = 77          occurs    = 00001
           scope    = GLOBAL
:recditem  name     = FILL
           level    = 77          occurs    = 00001
           scope    = GLOBAL
:recditem  name     = LOG_DATA
           level    = 77          occurs    = 00001
           scope    = GLOBAL
:recditem  name     = LOG_DATA_LENGTH
           level    = 77          occurs    = 00001
           scope    = GLOBAL
:recditem  name     = MODE_NAME
           level    = 77          occurs    = 00001
           scope    = GLOBAL
:recditem  name     = MODE_NAME_LENGTH
           level    = 77          occurs    = 00001
           scope    = GLOBAL
:recditem  name     = PARTNER_LU_NAME
           level    = 77          occurs    = 00001
           scope    = GLOBAL
:recditem  name     = PARTNER_LU_NAME_LENGTH
           level    = 77          occurs    = 00001
           scope    = GLOBAL
:recditem  name     = PREPARE_TO_RECEIVE_TYPE
           level    = 77          occurs    = 00001
           scope    = GLOBAL
:recditem  name     = RECEIVED_LENGTH
           level    = 77          occurs    = 00001
           scope    = GLOBAL
:recditem  name     = RECEIVE_TYPE
           level    = 77          occurs    = 00001
           scope    = GLOBAL
:recditem  name     = REQUESTED_LENGTH
           level    = 77          occurs    = 00001
           scope    = GLOBAL
:recditem  name     = REQUEST_TO_SEND_RECEIVED
           level    = 77          occurs    = 00001
           scope    = GLOBAL
:recditem  name     = RETURN_CONTROL
           level    = 77          occurs    = 00001
           scope    = GLOBAL
:recditem  name     = SEND_LENGTH
           level    = 77          occurs    = 00001
           scope    = GLOBAL
:recditem  name     = SEND_TYPE
           level    = 77          occurs    = 00001
           scope    = GLOBAL
:recditem  name     = STATUS_RECEIVED
           level    = 77          occurs    = 00001
           scope    = GLOBAL
:recditem  name     = SYNC_LEVEL
           level    = 77          occurs    = 00001
           scope    = GLOBAL
:recditem  name     = SYM_DEST_NAME
           level    = 77          occurs    = 00001
           scope    = GLOBAL
```

```
:recditem  name      = TP_NAME
           level     = 77         occurs    = 00001
           scope     = GLOBAL
:recditem  name      = TP_NAME_LENGTH
           level     = 77         occurs    = 00001
           scope     = GLOBAL
:erecord.
```

C Pseudonym File (CMC)

```
/**********************************************************************/
/*            CPI Communications Enumerated Constants                 */
/**********************************************************************/

     /*****************************************************/
     /* CM_INT32 should be a 32-bit, signed integer.  The */
     /* following #define is system dependent and may     */
     /* need to be changed on systems where signed long   */
     /* int does not define a 32-bit, signed integer.     */
     /*****************************************************/

#define CM_INT32 signed long int

     /*****************************************************/
     /* SYSTEM should be changed in the following #define */
     /* to one of the following values:   CM_VM, CM_OS2,  */
     /* CM_MVS, or CM_OS400.                              */
     /*                                                   */
     /* This is necessary for the proper setting of       */
     /* CM_ENTRY and CM_PTR below.                        */
     /*****************************************************/

#define SYSTEM

#ifdef CM_VM
#      define CM_ENTRY extern void
#      define CM_PTR *
#endif

#ifdef CM_OS2
#      define CM_ENTRY extern void pascal far _loadds
#      define CM_PTR far *
#endif

#ifdef CM_MVS
#      define CM_ENTRY extern void
#      define CM_PTR *
#endif

#ifdef CM_OS400
#      define CM_ENTRY extern void
#      define CM_PTR *
#endif

typedef CM_INT32 CM_CONVERSATION_STATE;
typedef CM_INT32 CM_CONVERSATION_TYPE;
typedef CM_INT32 CM_DATA_RECEIVED_TYPE;
typedef CM_INT32 CM_DEALLOCATE_TYPE;
typedef CM_INT32 CM_ERROR_DIRECTION;
typedef CM_INT32 CM_FILL;
typedef CM_INT32 CM_PREPARE_TO_RECEIVE_TYPE;
typedef CM_INT32 CM_RECEIVE_TYPE;
typedef CM_INT32 CM_REQUEST_TO_SEND_RECEIVED;
typedef CM_INT32 CM_RETURN_CODE;
typedef CM_INT32 CM_RETURN_CONTROL;
typedef CM_INT32 CM_SEND_TYPE;
typedef CM_INT32 CM_STATUS_RECEIVED;
typedef CM_INT32 CM_SYNC_LEVEL;


/*  conversation_state values  */

#define CM_INITIALIZE_STATE              (CM_CONVERSATION_STATE) 2
#define CM_SEND_STATE                    (CM_CONVERSATION_STATE) 3
#define CM_RECEIVE_STATE                 (CM_CONVERSATION_STATE) 4
#define CM_SEND_PENDING_STATE            (CM_CONVERSATION_STATE) 5
#define CM_CONFIRM_STATE                 (CM_CONVERSATION_STATE) 6
#define CM_CONFIRM_SEND_STATE            (CM_CONVERSATION_STATE) 7
#define CM_CONFIRM_DEALLOCATE_STATE      (CM_CONVERSATION_STATE) 8
#define CM_DEFER_RECEIVE_STATE           (CM_CONVERSATION_STATE) 9
#define CM_DEFER_DEALLOCATE_STATE        (CM_CONVERSATION_STATE) 10
#define CM_SYNC_POINT_STATE              (CM_CONVERSATION_STATE) 11
#define CM_SYNC_POINT_SEND_STATE         (CM_CONVERSATION_STATE) 12
#define CM_SYNC_POINT_DEALLOCATE_STATE   (CM_CONVERSATION_STATE) 13


/*  conversation_type values  */

#define CM_BASIC_CONVERSATION            (CM_CONVERSATION_TYPE) 0
#define CM_MAPPED_CONVERSATION           (CM_CONVERSATION_TYPE) 1


/*  data_received values  */

#define CM_NO_DATA_RECEIVED              (CM_DATA_RECEIVED_TYPE) 0
#define CM_DATA_RECEIVED                 (CM_DATA_RECEIVED_TYPE) 1
#define CM_COMPLETE_DATA_RECEIVED        (CM_DATA_RECEIVED_TYPE) 2
#define CM_INCOMPLETE_DATA_RECEIVED      (CM_DATA_RECEIVED_TYPE) 3


/*  deallocate_type values  */

#define CM_DEALLOCATE_SYNC_LEVEL         (CM_DEALLOCATE_TYPE) 0
#define CM_DEALLOCATE_FLUSH              (CM_DEALLOCATE_TYPE) 1
#define CM_DEALLOCATE_CONFIRM            (CM_DEALLOCATE_TYPE) 2
#define CM_DEALLOCATE_ABEND              (CM_DEALLOCATE_TYPE) 3


/*  error_direction values  */

#define CM_RECEIVE_ERROR                 (CM_ERROR_DIRECTION) 0
#define CM_SEND_ERROR                    (CM_ERROR_DIRECTION) 1


/*  fill values  */

#define CM_FILL_LL                       (CM_FILL) 0
#define CM_FILL_BUFFER                   (CM_FILL) 1


/*  prepare_to_receive_type values  */

#define CM_PREP_TO_RECEIVE_SYNC_LEVEL    (CM_PREPARE_TO_RECEIVE_TYPE) 0
#define CM_PREP_TO_RECEIVE_FLUSH         (CM_PREPARE_TO_RECEIVE_TYPE) 1
#define CM_PREP_TO_RECEIVE_CONFIRM       (CM_PREPARE_TO_RECEIVE_TYPE) 2
```

```
/*  receive_type values  */

#define CM_RECEIVE_AND_WAIT               (CM_RECEIVE_TYPE) 0
#define CM_RECEIVE_IMMEDIATE              (CM_RECEIVE_TYPE) 1


/*  request_to_send_received values  */

#define CM_REQ_TO_SEND_NOT_RECEIVED       (CM_REQUEST_TO_SEND_RECEIVED) 0
#define CM_REQ_TO_SEND_RECEIVED           (CM_REQUEST_TO_SEND_RECEIVED) 1


/*  return_code values  */

#define CM_OK                             (CM_RETURN_CODE) 0
#define CM_ALLOCATE_FAILURE_NO_RETRY      (CM_RETURN_CODE) 1
#define CM_ALLOCATE_FAILURE_RETRY         (CM_RETURN_CODE) 2
#define CM_CONVERSATION_TYPE_MISMATCH     (CM_RETURN_CODE) 3
#define CM_PIP_NOT_SPECIFIED_CORRECTLY    (CM_RETURN_CODE) 5
#define CM_SECURITY_NOT_VALID             (CM_RETURN_CODE) 6
#define CM_SYNC_LVL_NOT_SUPPORTED_LU      (CM_RETURN_CODE) 7
#define CM_SYNC_LVL_NOT_SUPPORTED_PGM     (CM_RETURN_CODE) 8
#define CM_TPN_NOT_RECOGNIZED             (CM_RETURN_CODE) 9
#define CM_TP_NOT_AVAILABLE_NO_RETRY      (CM_RETURN_CODE) 10
#define CM_TP_NOT_AVAILABLE_RETRY         (CM_RETURN_CODE) 11
#define CM_DEALLOCATED_ABEND              (CM_RETURN_CODE) 17
#define CM_DEALLOCATED_NORMAL             (CM_RETURN_CODE) 18
#define CM_PARAMETER_ERROR                (CM_RETURN_CODE) 19
#define CM_PRODUCT_SPECIFIC_ERROR         (CM_RETURN_CODE) 20
#define CM_PROGRAM_ERROR_NO_TRUNC         (CM_RETURN_CODE) 21
#define CM_PROGRAM_ERROR_PURGING          (CM_RETURN_CODE) 22
#define CM_PROGRAM_ERROR_TRUNC            (CM_RETURN_CODE) 23
#define CM_PROGRAM_PARAMETER_CHECK        (CM_RETURN_CODE) 24
#define CM_PROGRAM_STATE_CHECK            (CM_RETURN_CODE) 25
#define CM_RESOURCE_FAILURE_NO_RETRY      (CM_RETURN_CODE) 26
#define CM_RESOURCE_FAILURE_RETRY         (CM_RETURN_CODE) 27
#define CM_UNSUCCESSFUL                   (CM_RETURN_CODE) 28
#define CM_DEALLOCATED_ABEND_SVC          (CM_RETURN_CODE) 30
#define CM_DEALLOCATED_ABEND_TIMER        (CM_RETURN_CODE) 31
#define CM_SVC_ERROR_NO_TRUNC             (CM_RETURN_CODE) 32
#define CM_SVC_ERROR_PURGING              (CM_RETURN_CODE) 33
#define CM_SVC_ERROR_TRUNC                (CM_RETURN_CODE) 34
#define CM_TAKE_BACKOUT                   (CM_RETURN_CODE) 100
#define CM_DEALLOCATED_ABEND_BO           (CM_RETURN_CODE) 130
#define CM_DEALLOCATED_ABEND_SVC_BO       (CM_RETURN_CODE) 131
#define CM_DEALLOCATED_ABEND_TIMER_BO     (CM_RETURN_CODE) 132
#define CM_RESOURCE_FAIL_NO_RETRY_BO      (CM_RETURN_CODE) 133
#define CM_RESOURCE_FAILURE_RETRY_BO      (CM_RETURN_CODE) 134
#define CM_DEALLOCATED_NORMAL_BO          (CM_RETURN_CODE) 135


/*  return_control values  */

#define CM_WHEN_SESSION_ALLOCATED         (CM_RETURN_CONTROL) 0
#define CM_IMMEDIATE                      (CM_RETURN_CONTROL) 1
```

```
/*  send_type values  */

#define CM_BUFFER_DATA                   (CM_SEND_TYPE) 0
#define CM_SEND_AND_FLUSH                (CM_SEND_TYPE) 1
#define CM_SEND_AND_CONFIRM              (CM_SEND_TYPE) 2
#define CM_SEND_AND_PREP_TO_RECEIVE      (CM_SEND_TYPE) 3
#define CM_SEND_AND_DEALLOCATE           (CM_SEND_TYPE) 4


/*  status_received values  */

#define CM_NO_STATUS_RECEIVED            (CM_STATUS_RECEIVED) 0
#define CM_SEND_RECEIVED                 (CM_STATUS_RECEIVED) 1
#define CM_CONFIRM_RECEIVED              (CM_STATUS_RECEIVED) 2
#define CM_CONFIRM_SEND_RECEIVED         (CM_STATUS_RECEIVED) 3
#define CM_CONFIRM_DEALLOC_RECEIVED      (CM_STATUS_RECEIVED) 4
#define CM_TAKE_COMMIT                   (CM_STATUS_RECEIVED) 5
#define CM_TAKE_COMMIT_SEND              (CM_STATUS_RECEIVED) 6
#define CM_TAKE_COMMIT_DEALLOCATE        (CM_STATUS_RECEIVED) 7


/*  sync_level values  */

#define CM_NONE                          (CM_SYNC_LEVEL) 0
#define CM_CONFIRM                       (CM_SYNC_LEVEL) 1
#define CM_SYNC_POINT                    (CM_SYNC_LEVEL) 2

/**********************************************************************/
/*            CPI Communications routine prototypes                   */
/**********************************************************************/

CM_ENTRY cmaccp(unsigned char CM_PTR,   /* conversation_ID          */
                CM_INT32 CM_PTR);       /* return_code              */
CM_ENTRY cmallc(unsigned char CM_PTR,   /* conversation_ID          */
                CM_INT32 CM_PTR);       /* return_code              */
CM_ENTRY cmcfm(unsigned char CM_PTR,    /* conversation_ID          */
               CM_INT32 CM_PTR,         /* request_to_send_received */
               CM_INT32 CM_PTR);        /* return_code              */
CM_ENTRY cmcfmd(unsigned char CM_PTR,   /* conversation_ID          */
                CM_INT32 CM_PTR);       /* return_code              */
CM_ENTRY cmdeal(unsigned char CM_PTR,   /* conversation_ID          */
                CM_INT32 CM_PTR);       /* return_code              */
CM_ENTRY cmecs(unsigned char CM_PTR,    /* conversation_ID          */
               CM_INT32 CM_PTR,         /* conversation_state       */
               CM_INT32 CM_PTR);        /* return_code              */
CM_ENTRY cmect(unsigned char CM_PTR,    /* conversation_ID          */
               CM_INT32 CM_PTR,         /* conversation_type        */
               CM_INT32 CM_PTR);        /* return_code              */
CM_ENTRY cmemn(unsigned char CM_PTR,    /* conversation_ID          */
               unsigned char CM_PTR,    /* mode_name                */
               CM_INT32 CM_PTR,         /* mode_name_length         */
               CM_INT32 CM_PTR);        /* return_code              */
CM_ENTRY cmepln(unsigned char CM_PTR,   /* conversation_ID          */
                unsigned char CM_PTR,   /* partner_LU_name          */
                CM_INT32 CM_PTR,        /* partner_LU_name_length   */
                CM_INT32 CM_PTR);       /* return_code              */
```

```
CM_ENTRY cmesl(unsigned char CM_PTR,   /* conversation_ID          */
               CM_INT32 CM_PTR,        /* sync_level               */
               CM_INT32 CM_PTR);       /* return_code              */
CM_ENTRY cmflus(unsigned char CM_PTR,  /* conversation_ID          */
                CM_INT32 CM_PTR);      /* return_code              */
CM_ENTRY cminit(unsigned char CM_PTR,  /* conversation_ID          */
                unsigned char CM_PTR,  /* sym_dest_name            */
                CM_INT32 CM_PTR);      /* return_code              */
CM_ENTRY cmptr(unsigned char CM_PTR,   /* conversation_ID          */
               CM_INT32 CM_PTR);       /* return_code              */
CM_ENTRY cmrcv(unsigned char CM_PTR,   /* conversation_ID          */
               unsigned char CM_PTR,   /* buffer                   */
               CM_INT32 CM_PTR,        /* requested_length         */
               CM_INT32 CM_PTR,        /* data_received            */
               CM_INT32 CM_PTR,        /* received_length          */
               CM_INT32 CM_PTR,        /* status_received          */
               CM_INT32 CM_PTR,        /* request_to_send_received */
               CM_INT32 CM_PTR);       /* return_code              */
CM_ENTRY cmrts(unsigned char CM_PTR,   /* conversation_ID          */
               CM_INT32 CM_PTR);       /* return_code              */
CM_ENTRY cmsend(unsigned char CM_PTR,  /* conversation_ID          */
                unsigned char CM_PTR,  /* buffer                   */
                CM_INT32 CM_PTR,       /* send_length              */
                CM_INT32 CM_PTR,       /* request_to_send_received */
                CM_INT32 CM_PTR);      /* return_code              */
CM_ENTRY cmserr(unsigned char CM_PTR,  /* conversation_ID          */
                CM_INT32 CM_PTR,       /* request_to_send_received */
                CM_INT32 CM_PTR);      /* return_code              */
CM_ENTRY cmsct(unsigned char CM_PTR,   /* conversation_ID          */
               CM_INT32 CM_PTR,        /* conversation_type        */
               CM_INT32 CM_PTR);       /* return_code              */
CM_ENTRY cmsdt(unsigned char CM_PTR,   /* conversation_ID          */
               CM_INT32 CM_PTR,        /* deallocate_type          */
               CM_INT32 CM_PTR);       /* return_code              */
CM_ENTRY cmsed(unsigned char CM_PTR,   /* conversation_ID          */
               CM_INT32 CM_PTR,        /* error_direction          */
               CM_INT32 CM_PTR);       /* return_code              */
CM_ENTRY cmsf(unsigned char CM_PTR,    /* conversation_ID          */
              CM_INT32 CM_PTR,         /* fill                     */
              CM_INT32 CM_PTR);        /* return_code              */
CM_ENTRY cmsld(unsigned char CM_PTR,   /* conversation_ID          */
               unsigned char CM_PTR,   /* log_data                 */
               CM_INT32 CM_PTR,        /* log_data_length          */
               CM_INT32 CM_PTR);       /* return_code              */
CM_ENTRY cmsmn(unsigned char CM_PTR,   /* conversation_ID          */
               unsigned char CM_PTR,   /* mode_name                */
               CM_INT32 CM_PTR,        /* mode_name_length         */
               CM_INT32 CM_PTR);       /* return_code              */
CM_ENTRY cmspln(unsigned char CM_PTR,  /* conversation_ID          */
                unsigned char CM_PTR,  /* partner_LU_name          */
                CM_INT32 CM_PTR,       /* partner_LU_name_length   */
                CM_INT32 CM_PTR);      /* return_code              */
CM_ENTRY cmsptr(unsigned char CM_PTR,  /* conversation_ID          */
                CM_INT32 CM_PTR,       /* prepare_to_receive_type  */
                CM_INT32 CM_PTR);      /* return_code              */
```

```
CM_ENTRY cmsrt(unsigned char CM_PTR,    /* conversation_ID          */
               CM_INT32 CM_PTR,         /* receive_type             */
               CM_INT32 CM_PTR);        /* return_code              */
CM_ENTRY cmsrc(unsigned char CM_PTR,    /* conversation_ID          */
               CM_INT32 CM_PTR,         /* return_control           */
               CM_INT32 CM_PTR);        /* return_code              */
CM_ENTRY cmsst(unsigned char CM_PTR,    /* conversation_ID          */
               CM_INT32 CM_PTR,         /* send_type                */
               CM_INT32 CM_PTR);        /* return_code              */
CM_ENTRY cmssl(unsigned char CM_PTR,    /* conversation_ID          */
               CM_INT32 CM_PTR,         /* sync_level               */
               CM_INT32 CM_PTR);        /* return_code              */
CM_ENTRY cmstpn(unsigned char CM_PTR,   /* conversation_ID          */
                unsigned char CM_PTR,   /* TP_name                  */
                CM_INT32 CM_PTR,        /* TP_name_length           */
                CM_INT32 CM_PTR);       /* return_code              */
CM_ENTRY cmtrts(unsigned char CM_PTR,   /* conversation_ID          */
                CM_INT32 CM_PTR,        /* request_to_send_received */
                CM_INT32 CM_PTR);       /* return_code              */

/**********************************************************************/
/*    #pragma linkage directives                                      */
/**********************************************************************/
#if defined(CM_VM) || defined(CM_MVS) || defined(CM_OS400)
#     pragma linkage (cmaccp, OS)
#     pragma linkage (cmallc, OS)
#     pragma linkage (cmcfm,  OS)
#     pragma linkage (cmcfmd, OS)
#     pragma linkage (cmdeal, OS)
#     pragma linkage (cmecs,  OS)
#     pragma linkage (cmect,  OS)
#     pragma linkage (cmemn,  OS)
#     pragma linkage (cmepln, OS)
#     pragma linkage (cmesl,  OS)
#     pragma linkage (cmflus, OS)
#     pragma linkage (cminit, OS)
#     pragma linkage (cmptr,  OS)
#     pragma linkage (cmrcv,  OS)
#     pragma linkage (cmrts,  OS)
#     pragma linkage (cmsend, OS)
#     pragma linkage (cmserr, OS)
#     pragma linkage (cmsct,  OS)
#     pragma linkage (cmsdt,  OS)
#     pragma linkage (cmsed,  OS)
#     pragma linkage (cmsf,   OS)
#     pragma linkage (cmsld,  OS)
#     pragma linkage (cmsmn,  OS)
#     pragma linkage (cmspln, OS)
#     pragma linkage (cmsptr, OS)
#     pragma linkage (cmsrt,  OS)
#     pragma linkage (cmsrc,  OS)
#     pragma linkage (cmsst,  OS)
#     pragma linkage (cmssl,  OS)
#     pragma linkage (cmstpn, OS)
#     pragma linkage (cmtrts, OS)
#endif
```

COBOL Pseudonym File (CMCOBOL)

```
*COPY CMCOBOL
***************************************************
* NOTE: BUFFER MUST BE DEFINED IN WORKING STORAGE *
***************************************************
*
 01  CONVERSATION-ID            PIC X(8).
*
 01  CONVERSATION-STATE         PIC 9(9) COMP-4.
     88  CM-INITIALIZE-STATE             VALUE 2.
     88  CM-SEND-STATE                   VALUE 3.
     88  CM-RECEIVE-STATE                VALUE 4.
     88  CM-SEND-PENDING-STATE           VALUE 5.
     88  CM-CONFIRM-STATE                VALUE 6.
     88  CM-CONFIRM-SEND-STATE           VALUE 7.
     88  CM-CONFIRM-DEALLOCATE-STATE     VALUE 8.
     88  CM-DEFER-RECEIVE-STATE          VALUE 9.
     88  CM-DEFER-DEALLOCATE-STATE       VALUE 10.
     88  CM-SYNC-POINT-STATE             VALUE 11.
     88  CM-SYNC-POINT-SEND-STATE        VALUE 12.
     88  CM-SYNC-POINT-DEALLOCATE-STATE  VALUE 13.
*
 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-PIP-NOT-SPECIFIED-CORRECTLY   VALUE 5.
     88   CM-SECURITY-NOT-VALID            VALUE 6.
     88   CM-SYNC-LVL-NOT-SUPPORTED-LU     VALUE 7.
     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.
     88   CM-DEALLOCATED-ABEND-SVC         VALUE 30.
     88   CM-DEALLOCATED-ABEND-TIMER       VALUE 31.
     88   CM-SVC-ERROR-NO-TRUNC            VALUE 32.
     88   CM-SVC-ERROR-PURGING             VALUE 33.
     88   CM-SVC-ERROR-TRUNC               VALUE 34.
     88   CM-TAKE-BACKOUT                  VALUE 100.
     88   CM-DEALLOCATED-ABEND-BO          VALUE 130.
     88   CM-DEALLOCATED-ABEND-SVC-BO      VALUE 131.
     88   CM-DEALLOCATED-ABEND-TIMER-BO    VALUE 132.
     88   CM-RESOURCE-FAIL-NO-RETRY-BO     VALUE 133.
     88   CM-RESOURCE-FAILURE-RETRY-BO     VALUE 134.
     88   CM-DEALLOCATED-NORMAL-BO         VALUE 135.
*
 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.
     88  CM-TAKE-COMMIT               VALUE 5.
     88  CM-TAKE-COMMIT-SEND          VALUE 6.
     88  CM-TAKE-COMMIT-DEALLOCATE    VALUE 7.
*
 01  SYNC-LEVEL                 PIC 9(9) COMP-4.
     88   CM-NONE               VALUE 0.
     88   CM-CONFIRM            VALUE 1.
     88   CM-SYNC-POINT         VALUE 2.
*
 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.
```

FORTRAN Pseudonym File (CMFORTRN)

```
*COPY CMFORTRN
C*************** SAA CPI Communications Variable Names ****************
C*
C* Short and long variable names are included in this file.  Short names
C* are provided for migration purposes.
C*
C**********************************************************************

C *** conversation_state **************************************
C
        INTEGER CM_INITIALIZE_STATE           /2/
        INTEGER CM_SEND_STATE                 /3/
        INTEGER CM_RECEIVE_STATE              /4/
        INTEGER CM_SEND_PENDING_STATE         /5/
        INTEGER CM_CONFIRM_STATE              /6/
        INTEGER CM_CONFIRM_SEND_STATE         /7/
        INTEGER CM_CONFIRM_DEALLOCATE_STATE   /8/
        INTEGER CM_DEFER_RECEIVE_STATE        /9/
        INTEGER CM_DEFER_DEALLOCATE_STATE     /10/
        INTEGER CM_SYNC_POINT_STATE           /11/
        INTEGER CM_SYNC_POINT_SEND_STATE      /12/
        INTEGER CM_SYNC_POINT_DEALLOCATE_STATE /13/

        INTEGER INITST                        /2/
        INTEGER SENDST                        /3/
        INTEGER RCVST                         /4/
        INTEGER SPNDST                        /5/
        INTEGER CFMST                         /6/
        INTEGER CSNDST                        /7/
        INTEGER CDEAST                        /8/
        INTEGER DRCVST                        /9/
        INTEGER DDEAST                        /10/
        INTEGER SPST                          /11/
        INTEGER SPSST                         /12/
        INTEGER SPDST                         /13/

C *** conversation_type ***************************************
C
        INTEGER CM_BASIC_CONVERSATION  /0/
        INTEGER CM_MAPPED_CONVERSATION /1/

        INTEGER BASIC                  /0/
        INTEGER MAPPED                 /1/

C *** data_received *******************************************
C
        INTEGER CM_NO_DATA_RECEIVED         /0/
        INTEGER CM_DATA_RECEIVED            /1/
        INTEGER CM_COMPLETE_DATA_RECEIVED   /2/
        INTEGER CM_INCOMPLETE_DATA_RECEIVED /3/

        INTEGER NODATA                      /0/
        INTEGER DATREC                      /1/
        INTEGER COMDAT                      /2/
        INTEGER INCDAT                      /3/
```

```
C *** deallocate_type ******************************************
C
        INTEGER CM_DEALLOCATE_SYNC_LEVEL /0/
        INTEGER CM_DEALLOCATE_FLUSH      /1/
        INTEGER CM_DEALLOCATE_CONFIRM    /2/
        INTEGER CM_DEALLOCATE_ABEND      /3/

        INTEGER DESYNC                   /0/
        INTEGER DEFLUS                   /1/
        INTEGER DECONF                   /2/
        INTEGER DEABTY                   /3/


C *** error_direction ******************************************
C
        INTEGER CM_RECEIVE_ERROR /0/
        INTEGER CM_SEND_ERROR    /1/

        INTEGER RCVERR           /0/
        INTEGER SNDERR           /1/


C *** fill *****************************************************
C
        INTEGER CM_FILL_LL     /0/
        INTEGER CM_FILL_BUFFER /1/

        INTEGER FIL_LL         /0/
        INTEGER FILBUF         /1/


C *** prepare_to_receive_type **********************************
C
        INTEGER CM_PREP_TO_RECEIVE_SYNC_LEVEL /0/
        INTEGER CM_PREP_TO_RECEIVE_FLUSH      /1/
        INTEGER CM_PREP_TO_RECEIVE_CONFIRM    /2/

        INTEGER PTR_SL                        /0/
        INTEGER PTRFLS                        /1/
        INTEGER PTRCON                        /2/


C *** receive_type *********************************************
C
        INTEGER CM_RECEIVE_AND_WAIT  /0/
        INTEGER CM_RECEIVE_IMMEDIATE /1/

        INTEGER RCVWAT               /0/
        INTEGER RCVIMM               /1/


C *** request_to_send_received *********************************
C
        INTEGER CM_REQ_TO_SEND_NOT_RECEIVED     /0/
        INTEGER CM_REQ_TO_SEND_RECEIVED         /1/

        INTEGER RTSNOT                          /0/
        INTEGER RTSREC                          /1/
```

```
C *** return_code *********************************************
C
        INTEGER CM_OK                          /0/
        INTEGER CM_ALLOCATE_FAILURE_NO_RETRY   /1/
        INTEGER CM_ALLOCATE_FAILURE_RETRY      /2/
        INTEGER CM_CONVERSATION_TYPE_MISMATCH  /3/
        INTEGER CM_PIP_NOT_SPECIFIED_CORRECTLY /5/
        INTEGER CM_SECURITY_NOT_VALID          /6/
        INTEGER CM_SYNC_LVL_NOT_SUPPORTED_LU   /7/
        INTEGER CM_SYNC_LVL_NOT_SUPPORTED_PGM  /8/
        INTEGER CM_TPN_NOT_RECOGNIZED          /9/
        INTEGER CM_TP_NOT_AVAILABLE_NO_RETRY   /10/
        INTEGER CM_TP_NOT_AVAILABLE_RETRY      /11/
        INTEGER CM_DEALLOCATED_ABEND           /17/
        INTEGER CM_DEALLOCATED_NORMAL          /18/
        INTEGER CM_PARAMETER_ERROR             /19/
        INTEGER CM_PRODUCT_SPECIFIC_ERROR      /20/
        INTEGER CM_PROGRAM_ERROR_NO_TRUNC      /21/
        INTEGER CM_PROGRAM_ERROR_PURGING       /22/
        INTEGER CM_PROGRAM_ERROR_TRUNC         /23/
        INTEGER CM_PROGRAM_PARAMETER_CHECK     /24/
        INTEGER CM_PROGRAM_STATE_CHECK         /25/
        INTEGER CM_RESOURCE_FAILURE_NO_RETRY   /26/
        INTEGER CM_RESOURCE_FAILURE_RETRY      /27/
        INTEGER CM_UNSUCCESSFUL                /28/
        INTEGER CM_DEALLOCATED_ABEND_SVC       /30/
        INTEGER CM_DEALLOCATED_ABEND_TIMER     /31/
        INTEGER CM_SVC_ERROR_NO_TRUNC          /32/
        INTEGER CM_SVC_ERROR_PURGING           /33/
        INTEGER CM_SVC_ERROR_TRUNC             /34/
        INTEGER CM_TAKE_BACKOUT                /100/
        INTEGER CM_DEALLOCATED_ABEND_BO        /130/
        INTEGER CM_DEALLOCATED_ABEND_SVC_BO    /131/
        INTEGER CM_DEALLOCATED_ABEND_TIMER_BO  /132/
        INTEGER CM_RESOURCE_FAIL_NO_RETRY_BO   /133/
        INTEGER CM_RESOURCE_FAILURE_RETRY_BO   /134/
        INTEGER CM_DEALLOCATED_NORMAL_BO       /135/

        INTEGER ALFLNR                         /1/
        INTEGER ALFLRE                         /2/
        INTEGER CNVMIS                         /3/
        INTEGER PNSC                           /5/
        INTEGER SECNVL                         /6/
        INTEGER SL_NSL                         /7/
        INTEGER SL_NS                          /8/
        INTEGER TPNAME                         /9/
        INTEGER TPNORE                         /10/
        INTEGER TPRET                          /11/
        INTEGER DEABND                         /17/
        INTEGER DENORM                         /18/
        INTEGER PARERR                         /19/
        INTEGER PRODER                         /20/
        INTEGER PENOTR                         /21/
        INTEGER PEPURG                         /22/
        INTEGER PETRNC                         /23/
        INTEGER PEPCHK                         /24/
        INTEGER STACHK                         /25/
        INTEGER RFNORE                         /26/
        INTEGER RFRET                          /27/
        INTEGER UNSUCC                         /28/
        INTEGER DABSVC                         /30/
        INTEGER DABTIM                         /31/
        INTEGER SVCENT                         /32/
        INTEGER SVCEP                          /33/
        INTEGER SVCET                          /34/



        INTEGER TAKEBO                         /100/
        INTEGER DABBO                          /130/
        INTEGER DABSBO                         /131/
        INTEGER DABTBO                         /132/
        INTEGER RFNRBO                         /133/
        INTEGER RFRBO                          /134/
        INTEGER DNORBO                         /135/


C *** return_control *****************************************
C
        INTEGER CM_WHEN_SESSION_ALLOCATED /0/
        INTEGER CM_IMMEDIATE              /1/

        INTEGER SESALL                    /0/
        INTEGER IMMED                     /1/


C *** send_type **********************************************
C
        INTEGER CM_BUFFER_DATA              /0/
        INTEGER CM_SEND_AND_FLUSH           /1/
        INTEGER CM_SEND_AND_CONFIRM         /2/
        INTEGER CM_SEND_AND_PREP_TO_RECEIVE /3/
        INTEGER CM_SEND_AND_DEALLOCATE      /4/

        INTEGER BUFDAT                      /0/
        INTEGER SNDFLS                      /1/
        INTEGER SNDCNF                      /2/
        INTEGER SNDPTR                      /3/
        INTEGER SNDDEL                      /4/


C *** status_received ****************************************
C
        INTEGER CM_NO_STATUS_RECEIVED       /0/
        INTEGER CM_SEND_RECEIVED            /1/
        INTEGER CM_CONFIRM_RECEIVED         /2/
        INTEGER CM_CONFIRM_SEND_RECEIVED    /3/
        INTEGER CM_CONFIRM_DEALLOC_RECEIVED /4/
        INTEGER CM_TAKE_COMMIT              /5/
        INTEGER CM_TAKE_COMMIT_SEND         /6/
        INTEGER CM_TAKE_COMMIT_DEALLOCATE   /7/

        INTEGER NOSTAT                      /0/
        INTEGER SNDREC                      /1/
        INTEGER CONRCV                      /2/
        INTEGER CONSND                      /3/
        INTEGER CONDEL                      /4/
        INTEGER TAKEC                       /5/
        INTEGER TAKECS                      /6/
        INTEGER TAKECD                      /7/
```

```
C *** sync_level ***********************************************
C
        INTEGER CM_NONE       /0/
        INTEGER CM_CONFIRM    /1/
        INTEGER CM_SYNC_POINT /2/

        INTEGER NONE          /0/
        INTEGER CONFRM        /1/
        INTEGER SYNCPT        /2/


C************** CPI Communications routine prototypes *****************
C*                                                                    *
C* Following are prototypes for all the CPI Communications            *
C* routines.                                                          *
C*                                                                    *
C*********************************************************************

     EXTERNAL CMACCP
     EXTERNAL CMALLC
     EXTERNAL CMCFM
     EXTERNAL CMCFMD
     EXTERNAL CMDEAL
     EXTERNAL CMECS
     EXTERNAL CMECT
     EXTERNAL CMEMN
     EXTERNAL CMEPLN
     EXTERNAL CMESL
     EXTERNAL CMFLUS
     EXTERNAL CMINIT
     EXTERNAL CMPTR
     EXTERNAL CMRCV
     EXTERNAL CMRTS
     EXTERNAL CMSCT
     EXTERNAL CMSDT
     EXTERNAL CMSED
     EXTERNAL CMSEND
     EXTERNAL CMSERR
     EXTERNAL CMSF
     EXTERNAL CMSLD
     EXTERNAL CMSMN
     EXTERNAL CMSPLN
     EXTERNAL CMSPTR
     EXTERNAL CMSRC
     EXTERNAL CMSRT
     EXTERNAL CMSSL
     EXTERNAL CMSST
     EXTERNAL CMSTPN
     EXTERNAL CMTRTS
C************* End of CPI Communications routine prototypes ***********
```

PL/I Pseudonym File (CMPLI)

```
  /*******************************************************************/
  /*                                                                 */
  /* PL/I INCLUDE FILE FOR SAA CPI COMMUNICATIONS SUPPORT            */
  /*                                                                 */
  /*******************************************************************/
%skip(3);
  /*******************************************************************/
  /* Conversation State Values                                       */
  /*******************************************************************/
  DECLARE
    ( cm_initialize_state               INITIAL(2),
      cm_send_state                     INITIAL(3),
      cm_receive_state                  INITIAL(4),
      cm_send_pending_state             INITIAL(5),
      cm_confirm_state                  INITIAL(6),
      cm_confirm_send_state             INITIAL(7),
      cm_confirm_deallocate_state       INITIAL(8),
      cm_defer_receive_state            INITIAL(9),
      cm_defer_deallocate_state         INITIAL(10),
      cm_sync_point_state               INITIAL(11),
      cm_sync_point_send_state          INITIAL(12),
      cm_sync_point_deallocate_state    INITIAL(13))
                                        FIXED BINARY(31) STATIC;
%skip;
  /*******************************************************************/
  /* Conversation Type Values                                        */
  /*******************************************************************/
  DECLARE
    ( cm_basic_conversation             INITIAL(0),
      cm_mapped_conversation            INITIAL(1))
                                        FIXED BINARY(31) STATIC;
%skip;
  /*******************************************************************/
  /* Data Received Values                                            */
  /*******************************************************************/
  DECLARE
    ( cm_no_data_received               INITIAL(0),
      cm_data_received                  INITIAL(1),
      cm_complete_data_received         INITIAL(2),
      cm_incomplete_data_received       INITIAL(3))
                                        FIXED BINARY(31) STATIC;

%skip;
  /*******************************************************************/
  /* Deallocate Type Values                                          */
  /*******************************************************************/
  DECLARE
    ( cm_deallocate_sync_level          INITIAL(0),
      cm_deallocate_flush               INITIAL(1),
      cm_deallocate_confirm             INITIAL(2),
      cm_deallocate_abend               INITIAL(3))
                                        FIXED BINARY(31) STATIC;
```

```
%skip;
  /******************************************************************/
  /* Error Direction Values                                         */
  /******************************************************************/
  DECLARE
    ( cm_receive_error                 INITIAL(0),
      cm_send_error                    INITIAL(1))
                                       FIXED BINARY(31) STATIC;
%skip;
  /******************************************************************/
  /* Fill Values                                                    */
  /******************************************************************/
  DECLARE
    ( cm_fill_ll                       INITIAL(0),
      cm_fill_buffer                   INITIAL(1))
                                       FIXED BINARY(31) STATIC;
%skip;
  /******************************************************************/
  /* Prepare to Receive Type Values                                 */
  /******************************************************************/
  DECLARE
    ( cm_prep_to_receive_sync_level    INITIAL(0),
      cm_prep_to_receive_flush         INITIAL(1),
      cm_prep_to_receive_confirm       INITIAL(2))
                                       FIXED BINARY(31) STATIC;
%skip;
  /******************************************************************/
  /* Receive Type Values                                            */
  /******************************************************************/
  DECLARE
    ( cm_receive_and_wait              INITIAL(0),
      cm_receive_immediate             INITIAL(1))
                                       FIXED BINARY(31) STATIC;
%skip;
  /******************************************************************/
  /* Return Code Values                                             */
  /******************************************************************/
  DECLARE
    ( cm_ok                            INITIAL(0),
      cm_allocate_failure_no_retry     INITIAL(1),
      cm_allocate_failure_retry        INITIAL(2),
      cm_conversation_type_mismatch    INITIAL(3),
      cm_pip_not_specified_correctly   INITIAL(5),
      cm_security_not_valid            INITIAL(6),
      cm_sync_lvl_not_supported_lu     INITIAL(7),
      cm_sync_lvl_not_supported_pgm    INITIAL(8),
      cm_tpn_not_recognized            INITIAL(9),
      cm_tp_not_available_no_retry     INITIAL(10),
      cm_tp_not_available_retry        INITIAL(11),
      cm_deallocated_abend             INITIAL(17),
      cm_deallocated_normal            INITIAL(18),
      cm_parameter_error               INITIAL(19),
      cm_product_specific_error        INITIAL(20),
      cm_program_error_no_trunc        INITIAL(21),
      cm_program_error_purging         INITIAL(22),
      cm_program_error_trunc           INITIAL(23),
      cm_program_parameter_check       INITIAL(24),
      cm_program_state_check           INITIAL(25),
      cm_resource_failure_no_retry     INITIAL(26),
      cm_resource_failure_retry        INITIAL(27),
      cm_unsuccessful               INITIAL(28),
      cm_deallocated_abend_svc      INITIAL(30),
      cm_deallocated_abend_timer    INITIAL(31),
      cm_svc_error_no_trunc         INITIAL(32),
      cm_svc_error_purging          INITIAL(33),
      cm_svc_error_trunc            INITIAL(34),
      cm_take_backout               INITIAL(100),
      cm_deallocated_abend_bo       INITIAL(130),
      cm_deallocated_abend_svc_bo   INITIAL(131),
      cm_deallocated_abend_timer_bo INITIAL(132),
      cm_resource_fail_no_retry_bo  INITIAL(133),
      cm_resource_failure_retry_bo  INITIAL(134),
      cm_deallocated_normal_bo      INITIAL(135))
                                    FIXED BINARY(31) STATIC;

%skip;
  /**************************************************************/
  /* Return Control Values                                      */
  /**************************************************************/
  DECLARE
    ( cm_when_session_allocated     INITIAL(0),
      cm_immediate                  INITIAL(1))
                                    FIXED BINARY(31) STATIC;
%skip;
  /**************************************************************/
  /* Request To Send Received Values                            */
  /**************************************************************/
  DECLARE
    ( cm_req_to_send_not_received   INITIAL(0),
      cm_req_to_send_received       INITIAL(1))
                                    FIXED BINARY(31) STATIC;
%skip;
  /**************************************************************/
  /* Send Type Values                                           */
  /**************************************************************/
  DECLARE
    ( cm_buffer_data                INITIAL(0),
      cm_send_and_flush             INITIAL(1),
      cm_send_and_confirm           INITIAL(2),
      cm_send_and_prep_to_receive   INITIAL(3),
      cm_send_and_deallocate        INITIAL(4))
                                    FIXED BINARY(31) STATIC;
%skip;
  /**************************************************************/
  /* Status Received Values                                     */
  /**************************************************************/
  DECLARE
    ( cm_no_status_received         INITIAL(0),
      cm_send_received              INITIAL(1),
      cm_confirm_received           INITIAL(2),
      cm_confirm_send_received      INITIAL(3),
      cm_confirm_dealloc_received   INITIAL(4),
      cm_take_commit                INITIAL(5),
      cm_take_commit_send           INITIAL(6),
      cm_take_commit_deallocate     INITIAL(7))
                                    FIXED BINARY(31) STATIC;
```

```
%skip;
  /******************************************************************/
  /* Sync Level Values                                              */
  /******************************************************************/
  DECLARE
    ( cm_none                          INITIAL(0),
      cm_confirm                       INITIAL(1),
      cm_sync_point                    INITIAL(2))
                                       FIXED BINARY(31) STATIC;
%skip;
  /******************************************************************/
  /* SAA CPI Communications Accept Conversation Call                */
  /******************************************************************/
  DECLARE
    CMACCP ENTRY
      ( CHAR(8),                   /* Out - Conversation ID         */
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /******************************************************************/
  /* SAA CPI Communications Allocate Call                           */
  /******************************************************************/
  DECLARE
    CMALLC ENTRY
      ( CHAR(8),                   /* In  - Conversation ID         */
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /******************************************************************/
  /* SAA CPI Communications Confirm Call                            */
  /******************************************************************/
  DECLARE
    CMCFM ENTRY
      ( CHAR(8),                   /* In  - Conversation ID         */
        FIXED BINARY(31),          /* Out - Request to Send Received*/
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /******************************************************************/
  /* SAA CPI Communications Confirmed Call                          */
  /******************************************************************/
  DECLARE
    CMCFMD ENTRY
      ( CHAR(8),                   /* In  - Conversation ID         */
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /******************************************************************/
  /* SAA CPI Communications Deallocate Call                         */
  /******************************************************************/
  DECLARE
    CMDEAL ENTRY
      ( CHAR(8),                   /* In  - Conversation ID         */
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
```

```
%skip;
  /******************************************************************/
  /* SAA CPI Communications Extract_Conversation_State Call         */
  /******************************************************************/
  DECLARE
    CMECS ENTRY
      ( CHAR(8),                   /* In  - Conversation ID         */
        FIXED BINARY(31),          /* Out - Conversation State      */
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /******************************************************************/
  /* SAA CPI Communications Extract_Conversation_Type Call          */
  /******************************************************************/
  DECLARE
    CMECT ENTRY
      ( CHAR(8),                   /* In  - Conversation ID         */
        FIXED BINARY(31),          /* Out - Conversation Type       */
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /******************************************************************/
  /* SAA CPI Communications Extract_Mode_Name Call                  */
  /******************************************************************/
  DECLARE
    CMEMN ENTRY
      ( CHAR(8),                   /* In  - Conversation ID         */
        CHAR(8),                   /* Out - Mode Name               */
        FIXED BINARY(31),          /* Out - Mode Name Length        */
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /******************************************************************/
  /* SAA CPI Communications Extract_Partner_LU_Name Call            */
  /******************************************************************/
  DECLARE
    CMEPLN ENTRY
      ( CHAR(8),                   /* In  - Conversation ID         */
        CHAR(17),                  /* Out - Partner LU Name         */
        FIXED BINARY(31),          /* Out - Partner LU Name Length  */
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /******************************************************************/
  /* SAA CPI Communications Extract_Sync_Level Call                 */
  /******************************************************************/
  DECLARE
    CMESL ENTRY
      ( CHAR(8),                   /* In  - Conversation ID         */
        FIXED BINARY(31),          /* Out - Sync Level              */
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
```

```
%skip;
  /******************************************************************/
  /* SAA CPI Communications Flush Call                              */
  /******************************************************************/
  DECLARE
    CMFLUS ENTRY
      ( CHAR(8),                   /* In  - Conversation ID         */
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /******************************************************************/
  /* SAA CPI Communications Initialize Conversation Call            */
  /******************************************************************/
  DECLARE
    CMINIT ENTRY
      ( CHAR(8),                   /* Out - Conversation ID         */
        CHAR(8),                   /* In  - Symbolic Dest Name      */
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /******************************************************************/
  /* SAA CPI Communications Prepare to Receive Call                 */
  /******************************************************************/
  DECLARE
    CMPTR ENTRY
      ( CHAR(8),                   /* In  - Conversation ID         */
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /******************************************************************/
  /* SAA CPI Communications Receive Call                            */
  /******************************************************************/
  DECLARE
    CMRCV ENTRY
      ( CHAR(8),                   /* In  - Conversation ID         */
        CHAR(*),                   /* Out - Buffer                  */
        FIXED BINARY(31),          /* In  - Requested Length        */
        FIXED BINARY(31),          /* Out - Data Received           */
        FIXED BINARY(31),          /* Out - Received Length         */
        FIXED BINARY(31),          /* Out - Status Received         */
        FIXED BINARY(31),          /* Out - Request To Send Received*/
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /******************************************************************/
  /* SAA CPI Communications Request to Send Call                    */
  /******************************************************************/
  DECLARE
    CMRTS ENTRY
      ( CHAR(8),                   /* In  - Conversation ID         */
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
```

```
%skip;
  /******************************************************************/
  /* SAA CPI Communications Send Data Call                          */
  /******************************************************************/
  DECLARE
    CMSEND ENTRY
      ( CHAR(8),                   /* In  - Conversation ID         */
        CHAR(*),                   /* In  - Buffer                  */
        FIXED BINARY(31),          /* In  - Send Length             */
        FIXED BINARY(31),          /* Out - Request To Send Received*/
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /******************************************************************/
  /* SAA CPI Communications Send Error Call                         */
  /******************************************************************/
  DECLARE
    CMSERR ENTRY
      ( CHAR(8),                   /* In  - Conversation ID         */
        FIXED BINARY(31),          /* Out - Request To Send Received*/
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /******************************************************************/
  /* SAA CPI Communications Set Conversation Type Call              */
  /******************************************************************/
  DECLARE
    CMSCT ENTRY
      ( CHAR(8),                   /* In  - Conversation ID         */
        FIXED BINARY(31),          /* In  - Conversation Type       */
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /******************************************************************/
  /* SAA CPI Communications Set Deallocate Type Call                */
  /******************************************************************/
  DECLARE
    CMSDT ENTRY
      ( CHAR(8),                   /* In  - Conversation ID         */
        FIXED BINARY(31),          /* In  - Deallocate Type         */
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /******************************************************************/
  /* SAA CPI Communications Set Error Direction Call                */
  /******************************************************************/
  DECLARE
    CMSED ENTRY
      ( CHAR(8),                   /* In  - Conversation ID         */
        FIXED BINARY(31),          /* In  - Error Direction         */
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
```

```
%skip;
  /****************************************************************/
  /* SAA CPI Communications Set Fill Call                         */
  /****************************************************************/
  DECLARE
    CMSF ENTRY
      ( CHAR(8),                  /* In  - Conversation ID        */
        FIXED BINARY(31),         /* In  - Fill                   */
        FIXED BINARY(31) )        /* Out - Return Code            */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /****************************************************************/
  /* SAA CPI Communications Set Log Data Call                     */
  /****************************************************************/
  DECLARE
    CMSLD ENTRY
      ( CHAR(8),                  /* In  - Conversation ID        */
        CHAR(*),                  /* In  - Log Data               */
        FIXED BINARY(31),         /* In  - Log Data Length        */
        FIXED BINARY(31) )        /* Out - Return Code            */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /****************************************************************/
  /* SAA CPI Communications Set Mode Name Call                    */
  /****************************************************************/
  DECLARE
    CMSMN ENTRY
      ( CHAR(8),                  /* In  - Conversation ID        */
        CHAR(8),                  /* In  - Mode Name              */
        FIXED BINARY(31),         /* In  - Mode Name Length       */
        FIXED BINARY(31) )        /* Out - Return Code            */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /****************************************************************/
  /* SAA CPI Communications Set Partner LU Name Call              */
  /****************************************************************/
  DECLARE
    CMSPLN ENTRY
      ( CHAR(8),                  /* In  - Conversation ID        */
        CHAR(17),                 /* In  - Partner LU Name        */
        FIXED BINARY(31),         /* In  - Partner LU Name Length */
        FIXED BINARY(31) )        /* Out - Return Code            */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /****************************************************************/
  /* SAA CPI Communications Set Prepare To Receive Type Call      */
  /****************************************************************/
  DECLARE
    CMSPTR ENTRY
      ( CHAR(8),                  /* In  - Conversation ID        */
        FIXED BINARY(31),         /* In  - Prepare To Receive Type */
        FIXED BINARY(31) )        /* Out - Return Code            */
      EXTERNAL OPTIONS(ASM INTER);
```

```
%skip;
  /*******************************************************************/
  /* SAA CPI Communications Set Receive Type Call                    */
  /*******************************************************************/
  DECLARE
    CMSRT ENTRY
      ( CHAR(8),                   /* In  - Conversation ID        */
        FIXED BINARY(31),          /* In  - Receive Type           */
        FIXED BINARY(31) )         /* Out - Return Code            */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /*******************************************************************/
  /* SAA CPI Communications Set Return Control Call                  */
  /*******************************************************************/
  DECLARE
    CMSRC ENTRY
      ( CHAR(8),                   /* In  - Conversation ID        */
        FIXED BINARY(31),          /* In  - Return Control         */
        FIXED BINARY(31) )         /* Out - Return Code            */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /*******************************************************************/
  /* SAA CPI Communications Set Send Type Call                       */
  /*******************************************************************/
  DECLARE
    CMSST ENTRY
      ( CHAR(8),                   /* In  - Conversation ID        */
        FIXED BINARY(31),          /* In  - Send Type              */
        FIXED BINARY(31) )         /* Out - Return Code            */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /*******************************************************************/
  /* SAA CPI Communications Set Sync Level Call                      */
  /*******************************************************************/
  DECLARE
    CMSSL ENTRY
      ( CHAR(8),                   /* In  - Conversation ID        */
        FIXED BINARY(31),          /* In  - Sync Level             */
        FIXED BINARY(31) )         /* Out - Return Code            */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /*******************************************************************/
  /* SAA CPI Communications Set TP Name Call                         */
  /*******************************************************************/
  DECLARE
    CMSTPN ENTRY
      ( CHAR(8),                   /* In  - Conversation ID        */
        CHAR(64),                  /* In  - TP Name                */
        FIXED BINARY(31),          /* In  - TP Name Length         */
        FIXED BINARY(31) )         /* Out - Return Code            */
      EXTERNAL OPTIONS(ASM INTER);
```

```
%skip;
  /******************************************************************/
  /* SAA CPI Communications Test Request To Send Received Call      */
  /******************************************************************/
  DECLARE
    CMTRTS ENTRY
      ( CHAR(8),                   /* In  - Conversation ID         */
        FIXED BINARY(31),          /* Out - Request To Send Received*/
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
```

REXX Pseudonym File (CMREXX)

```
                                       /* conversation_state          */
CM_INITIALIZE_STATE             = 2
CM_SEND_STATE                   = 3
CM_RECEIVE_STATE                = 4
CM_SEND_PENDING_STATE           = 5
CM_CONFIRM_STATE                = 6
CM_CONFIRM_SEND_STATE           = 7
CM_CONFIRM_DEALLOCATE_STATE     = 8
CM_DEFER_RECEIVE_STATE          = 9
CM_DEFER_DEALLOCATE_STATE       = 10
CM_SYNC_POINT_STATE             = 11
CM_SYNC_POINT_SEND_STATE        = 12
CM_SYNC_POINT_DEALLOCATE_STATE  = 13
                                       /* 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
                                       /* return_code                 */
CM_OK                           = 0
CM_ALLOCATE_FAILURE_NO_RETRY    = 1
CM_ALLOCATE_FAILURE_RETRY       = 2
CM_CONVERSATION_TYPE_MISMATCH   = 3
CM_PIP_NOT_SPECIFIED_CORRECTLY  = 5
CM_SECURITY_NOT_VALID           = 6
CM_SYNC_LVL_NOT_SUPPORTED_LU    = 7
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
CM_DEALLOCATED_ABEND_SVC        = 30
CM_DEALLOCATED_ABEND_TIMER      = 31
CM_SVC_ERROR_NO_TRUNC           = 32
CM_SVC_ERROR_PURGING            = 33
CM_SVC_ERROR_TRUNC              = 34
CM_TAKE_BACKOUT                 = 100
CM_DEALLOCATED_ABEND_BO         = 130
CM_DEALLOCATED_ABEND_SVC_BO     = 131
CM_DEALLOCATED_ABEND_TIMER_BO   = 132
CM_RESOURCE_FAIL_NO_RETRY_BO    = 133
CM_RESOURCE_FAILURE_RETRY_BO    = 134
CM_DEALLOCATED_NORMAL_BO        = 135
                                       /* 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
CM_TAKE_COMMIT                  = 5
CM_TAKE_COMMIT_SEND             = 6
CM_TAKE_COMMIT_DEALLOCATE       = 7

CM_NONE                         = 0    /* sync_level                  */
CM_CONFIRM                      = 1
CM_SYNC_POINT                   = 2
```

RPG Pseudonym File (CMRPG)

```
I*
I* RPG INCLUDE FOR SAA COMMUNICATIONS SUPPORT
I*
ICMCONS     DS
I*****************************************************************
I* conversation_state values:
I*
I*   CM_INITIALIZE_STATE             -- VALUE 2   (INITST)
I*   CM_SEND_STATE                   -- VALUE 3   (SENDST)
I*   CM_RECEIVE_STATE                -- VALUE 4   (RCVST)
I*   CM_SEND_PENDING_STATE           -- VALUE 5   (SPNDST)
I*   CM_CONFIRM_STATE                -- VALUE 6   (CFMST)
I*   CM_CONFIRM_SEND_STATE           -- VALUE 7   (CSNDST)
I*   CM_CONFIRM_DEALLOCATE_STATE     -- VALUE 8   (CDEAST)
I*   CM_DEFER_RECEIVE_STATE          -- VALUE 9   (DRCVST)
I*   CM_DEFER_DEALLOCATE_STATE       -- VALUE 10  (DDEAST)
I*   CM_SYNC_POINT_STATE             -- VALUE 11  (SPST)
I*   CM_SYNC_POINT_SEND_STATE        -- VALUE 12  (SPSST)
I*   CM_SYNC_POINT_DEALLOCATE_STATE  -- VALUE 13  (SPDST)
I*
I             2                     C         INITST
I             3                     C         SENDST
I             4                     C         RCVST
I             5                     C         SPNDST
I             6                     C         CFMST
I             7                     C         CSNDST
I             8                     C         CDEAST
I             9                     C         DRCVST
I             10                    C         DDEAST
I             11                    C         SPST
I             12                    C         SPSST
I             13                    C         SPDST
I*****************************************************************
I* conversation_type values:
I*
I*   CM_BASIC_CONVERSATION           -- VALUE 0   (BASIC)
I*   CM_MAPPED_CONVERSATION          -- VALUE 1   (MAPPED)
I*
I             0                     C         BASIC
I             1                     C         MAPPED
I*****************************************************************
I* data_received values:
I*
I*   CM_NO_DATA_RECEIVED             -- VALUE 0   (NODATA)
I*   CM_DATA_RECEIVED                -- VALUE 1   (DATREC)
I*   CM_COMPLETE_DATA_RECEIVED       -- VALUE 2   (COMDAT)
I*   CM_INCOMPLETE_DATA_RECEIVED     -- VALUE 3   (INCDAT)
I*
I             0                     C         NODATA
I             1                     C         DATREC
I             2                     C         COMDAT
I             3                     C         INCDAT
I*****************************************************************
I* deallocate_type values:
I*
I*   CM_DEALLOCATE_SYNC_LEVEL        -- VALUE 0   (DESYNC)
I*   CM_DEALLOCATE_FLUSH             -- VALUE 1   (DEFLUS)
```

```
I*   CM_DEALLOCATE_CONFIRM            -- VALUE 2   (DECONF)
I*   CM_DEALLOCATE_ABEND              -- VALUE 3   (DEABTY)
I*
I              0                     C         DESYNC
I              1                     C         DEFLUS
I              2                     C         DECONF
I              3                     C         DEABTY
I******************************************************************
I* error_direction values:
I*
I*   CM_RECEIVE_ERROR                 -- VALUE 0   (RCVERR)
I*   CM_SEND_ERROR                    -- VALUE 1   (SNDERR)
I*
I              0                     C         RCVERR
I              1                     C         SNDERR
I******************************************************************
I* fill values:
I*
I*   CM_FILL_LL                       -- VALUE 0   (FILLL)
I*   CM_FILL_BUFFER                   -- VALUE 1   (FILBUF)
I*
I              0                     C         FILLL
I              1                     C         FILBUF
I******************************************************************
I* prepare_to_receive_type values:
I*
I*   CM_PREP_TO_RECEIVE_SYNC_LEVEL    -- VALUE 0   (PTRSL)
I*   CM_PREP_TO_RECEIVE_FLUSH         -- VALUE 1   (PTRFLS)
I*   CM_PREP_TO_RECEIVE_CONFIRM       -- VALUE 2   (PTRCON)
I*
I              0                     C         PTRSL
I              1                     C         PTRFLS
I              2                     C         PTRCON
I******************************************************************
I* receive_type values:
I*
I*   CM_RECEIVE_AND_WAIT              -- VALUE 0   (RCVWAT)
I*   CM_RECEIVE_IMMEDIATE             -- VALUE 1   (RCVIMM)
I*
I              0                     C         RCVWAT
I              1                     C         RCVIMM
I******************************************************************
I* request_to_send_received values:
I*
I*   CM_REQ_TO_SEND_NOT_RECEIVED      -- VALUE 0   (RTSNOT)
I*   CM_REQ_TO_SEND_RECEIVED          -- VALUE 1   (RTSREC)
I*
I              0                     C         RTSNOT
I              1                     C         RTSREC
I******************************************************************
I* return_code values:
I*
I*   CM_OK                            -- VALUE 0   (CMOK)
I*   CM_ALLOCATE_FAILURE_NO_RETRY     -- VALUE 1   (ALFLNR)
I*   CM_ALLOCATE_FAILURE_RETRY        -- VALUE 2   (ALFLRE)
I*   CM_CONVERSATION_TYPE_MISMATCH    -- VALUE 3   (CNVMIS)
I*   CM_PIP_NOT_SPECIFIED_CORRECTLY   -- VALUE 5   (PIPNSC)
I*   CM_SECURITY_NOT_VALID            -- VALUE 6   (SECNVL)
I*   CM_SYNC_LVL_NOT_SUPPORTED_LU     -- VALUE 7   (SLNSLU)
I*   CM_SYNC_LVL_NOT_SUPPORTED_PGM    -- VALUE 8   (SLNSP)
```

```
I*   CM_TPN_NOT_RECOGNIZED            -- VALUE 9   (TPNAME)
I*   CM_TP_NOT_AVAILABLE_NO_RETRY     -- VALUE 10  (TPNORE)
I*   CM_TP_NOT_AVAILABLE_RETRY        -- VALUE 11  (TPRET)
I*   CM_DEALLOCATED_ABEND             -- VALUE 17  (DEABND)
I*   CM_DEALLOCATED_NORMAL            -- VALUE 18  (DENORM)
I*   CM_PARAMETER_ERROR               -- VALUE 19  (PARERR)
I*   CM_PRODUCT_SPECIFIC_ERROR        -- VALUE 20  (PRODER)
I*   CM_PROGRAM_ERROR_NO_TRUNC        -- VALUE 21  (PENOTR)
I*   CM_PROGRAM_ERROR_PURGING         -- VALUE 22  (PEPURG)
I*   CM_PROGRAM_ERROR_TRUNC           -- VALUE 23  (PETRNC)
I*   CM_PROGRAM_PARAMETER_CHECK       -- VALUE 24  (PEPCHK)
I*   CM_PROGRAM_STATE_CHECK           -- VALUE 25  (STACHK)
I*   CM_RESOURCE_FAILURE_NO_RETRY     -- VALUE 26  (RFNORE)
I*   CM_RESOURCE_FAILURE_RETRY        -- VALUE 27  (RFRET)
I*   CM_UNSUCCESSFUL                  -- VALUE 28  (UNSUCC)
I*   CM_DEALLOCATED_ABEND_SVC         -- VALUE 30  (DABSVC)
I*   CM_DEALLOCATED_ABEND_TIMER       -- VALUE 31  (DABTIM)
I*   CM_SVC_ERROR_NO_TRUNC            -- VALUE 32  (SVCENT)
I*   CM_SVC_ERROR_PURGING             -- VALUE 33  (SVCEP)
I*   CM_SVC_ERROR_TRUNC               -- VALUE 34  (SVCET)
I*   CM_TAKE_BACKOUT                  -- VALUE 100 (TAKEBO)
I*   CM_DEALLOCATED_ABEND_BO          -- VALUE 130 (DABBO)
I*   CM_DEALLOCATED_ABEND_SVC_BO      -- VALUE 131 (DABSBO)
I*   CM_DEALLOCATED_ABEND_TIMER_BO    -- VALUE 132 (DABTBO)
I*   CM_RESOURCE_FAIL_NO_RETRY_BO     -- VALUE 133 (RFNRBO)
I*   CM_RESOURCE_FAILURE_RETRY_BO     -- VALUE 134 (RFRBO)
I*   CM_DEALLOCATED_NORMAL_BO         -- VALUE 135 (DNORBO)
I*
I              0                     C         CMOK
I              1                     C         ALFLNR
I              2                     C         ALFLRE
I              3                     C         CNVMIS
I              5                     C         PIPNSC
I              6                     C         SECNVL
I              7                     C         SLNSLU
I              8                     C         SLNSP
I              9                     C         TPNAME
I              10                    C         TPNORE
I              11                    C         TPRET
I              17                    C         DEABND
I              18                    C         DENORM
I              19                    C         PARERR
I              20                    C         PRODER
I              21                    C         PENOTR
I              22                    C         PEPURG
I              23                    C         PETRNC
I              24                    C         PEPCHK
I              25                    C         STACHK
I              26                    C         RFNORE
I              27                    C         RFRET
I              28                    C         UNSUCC
I              30                    C         DABSVC
I              31                    C         DABTIM
I              32                    C         SVCENT
I              33                    C         SVCEP
I              34                    C         SVCET
I              100                   C         TAKEBO
I              130                   C         DABBO
I              131                   C         DABSBO
I              132                   C         DABTBO
```

```
I              133                  C         RFNRBO
I              134                  C         RFRBO
I              135                  C         DNORBO
I******************************************************************
I* return_control values:
I*
I*   CM_WHEN_SESSION_ALLOCATED        -- VALUE 0   (SESALL)
I*   CM_IMMEDIATE                     -- VALUE 1   (IMMED)
I*
I              0                    C         SESALL
I              1                    C         IMMED
I******************************************************************
I* send_type values:
I*
I*   CM_BUFFER_DATA                   -- VALUE 0   (BUFDAT)
I*   CM_SEND_AND_FLUSH                -- VALUE 1   (SNDFLS)
I*   CM_SEND_AND_CONFIRM              -- VALUE 2   (SNDCNF)
I*   CM_SEND_AND_PREP_TO_RECEIVE      -- VALUE 3   (SNDPTR)
I*   CM_SEND_AND_DEALLOCATE           -- VALUE 4   (SNDDEL)
I*
I              0                    C         BUFDAT
I              1                    C         SNDFLS
I              2                    C         SNDCNF
I              3                    C         SNDPTR
I              4                    C         SNDDEL
I******************************************************************
I* status_received values:
I*
I*   CM_NO_STATUS_RECEIVED            -- VALUE 0   (NOSTAT)
I*   CM_SEND_RECEIVED                 -- VALUE 1   (SNDREC)
I*   CM_CONFIRM_RECEIVED              -- VALUE 2   (CONRCV)
I*   CM_CONFIRM_SEND_RECEIVED         -- VALUE 3   (CONSND)
I*   CM_CONFIRM_DEALLOC_RECEIVED      -- VALUE 4   (CONDEL)
I*   CM_TAKE_COMMIT                   -- VALUE 5   (TAKEC)
I*   CM_TAKE_COMMIT_SEND              -- VALUE 6   (TAKECS)
I*   CM_TAKE_COMMIT_DEALLOCATE        -- VALUE 7   (TAKECD)
I*
I              0                    C         NOSTAT
I              1                    C         SNDREC
I              2                    C         CONRCV
I              3                    C         CONSND
I              4                    C         CONDEL
I              5                    C         TAKEC
I              6                    C         TAKECS
I              7                    C         TAKECD
I******************************************************************
I* sync_level values:
I*
I*   CM_NONE                          -- VALUE 0   (NONE)
I*   CM_CONFIRM                       -- VALUE 1   (CONFRM)
I*   CM_SYNC_POINT                    -- VALUE 2   (SYNCPT)
I*
I              0                    C         NONE
I              1                    C         CONFRM
I              2                    C         SYNCPT
```

Summary of Changes

The following changes have been made to this book since its last publication:

- All generic references to "synchronization point services" have been changed to refer to the SAA resource recovery interface.
- Two new examples have been added to Chapter 3, "Program-to-Program Communication Tutorial" that show how to use CPI Communications with the resource recovery interface.
- Support has been added for the CICS, IMS, MVS, OS/2, and OS/400 operating environments. This support is reflected in the body of the book and in the product appendixes that describe these implementations of CPI Communications.
- The COBOL pseudonym file has been moved to Appendix L, "Pseudonym Files." Also, a new pseudonym file for RPG has been added to this appendix.

Changes and additions 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.

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.

mode name. Part of the CPI Communications side information. The mode name is used by LU 6.2 to designate the properties for the session that will be allocated for a conversation.

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.

protected resource. A local or distributed resource that is updated in a synchronized manner during processing managed by the SAA resource recovery interface and a sync point manager.

R

remote program. The program at the other end of a conversation with respect to the reference program. Contrast with local program.

resource recovery interface. The SAA common programming interface to services and facilities that use two-phase commit protocols to coordinate changes to distributed resources.

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.

synchronization point. A reference point during transaction processing to which resources can be restored if a failure occurs.

sync point manager. A component of the operating environment that coordinates commit and backout processing among all the protected resources involved in a sync point transaction.

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

Reader's Comments

Systems Application Architecture
Common Programming Interface
Communications Reference

Publication No. SC26-4399-3

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

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

Be sure to print your name and address below if you would like a reply. If we have questions about your comment, may we call you? If so, please include your phone number.

Name

Address

Company or Organization

Phone No.

SC26-4399-3

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
Information Development
Department F14
PO BOX 12195
RESEARCH TRIANGLE PARK, NORTH CAROLINA, 27709-9990

Fold and tape | Please Do Not Staple | Fold and tape

IBM ®

Reader's Comments

Systems Application Architecture
Common Programming Interface
Communications Reference

Publication No. SC26-4399-3

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

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

Be sure to print your name and address below if you would like a reply. If we have questions about your comment, may we call you? If so, please include your phone number.

Name	Address
Company or Organization	
Phone No.	

SC26-4399-3

Reader's Comment Form

Fold and tape | Please Do Not Staple | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATE

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Information Development
Department F14
PO BOX 12195
RESEARCH TRIANGLE PARK, NORTH CAROLINA, 27709-9990

Fold and tape | Please Do Not Staple | Fold and tape

IBM®

File Number: S370-40

Printed in Denmark by N. Olaf Møller

Systems Application Architecture Library

GC26-4341	Overview
GC26-4675	Common Programming Interface: Summary
GC31-6810	Common Communications Support: Summary
SC26-4582	Common User Access: Advanced Interface Design Guide
SC26-4583	Common User Access: Basic Interface Design Guide
SC26-4362	Writing Applications: A Design Guide
GC26-4531	AD/Cycle Concepts
GC23-0576	An Introduction to SystemView

Common Programming Interface:

SC26-4355	Application Generator Reference
SC09-1308	C Reference – Level 2
SC26-4354	COBOL Reference
SC26-4399	Communications Reference
SC26-4348	Database Reference
SC26-4356	Dialog Reference
SC26-4357	FORTRAN Reference
SC26-4381	PL/I Reference
SC26-4359	Presentation Reference
SC26-4358	Procedures Language Reference
SC24-5549	Procedures Language Level 2 Reference
SC26-4349	Query Reference
SC26-4684	Repository Reference
SC31-6821	Resource Recovery Reference
SC09-1286	RPG Reference

SC26-4399-03