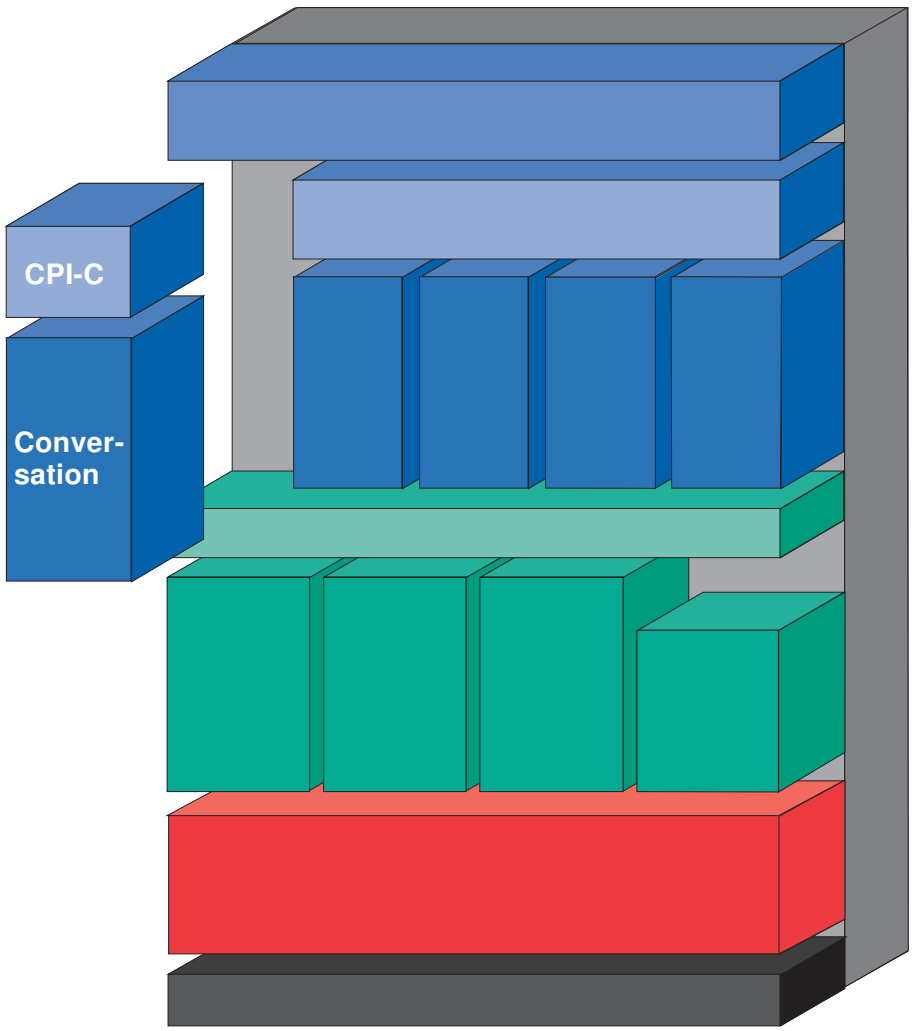
Common Programming Interface Communications Specification

SC31-6180-00



Common Programming Interface Communications Specification

SC31-6180-00

Note

Before using this document, read the general information under “Notices” on page xi.

First Edition (March 1993)

| This edition applies to the Common Programming Interface Communications (CPI-C) architecture.

| This edition incorporates text which is copyright 1990 X/Open Company Limited related to the following calls:

| • Cancel_Conversation (CMCANC)
| • Convert_Incoming (CMCNVI)
| • Convert_Outgoing (CMCNVO)
| • Extract_Security_User_ID (CMESUI)
| • Extract_TP_Name (CMETPN)
| • Set_Conversation_Security_Password (CMSCSP)
| • Set_Conversation_Security_Type (CMSCST)
| • Set_Conversation_Security_User_ID (CMSCSU)
| • Set_Processing_Mode (CMSPM)
| • Specify_Local_TP_Name (CMSLTP)
| • Wait_For_Conversation (CMWAIT)

| The information in the seventh edition of the *Systems Application Architecture Common Programming Interface: Communications Reference*, SC26-4399-06, was used as a base for this book. Therefore, only information that is new or technically different in this book is indicated by a vertical bar to the left of the change. Editorial changes that have no technical significance are not noted. See “Summary of Changes” on page 309 for a list of new and changed topics.

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 27709-9990
U.S.A.

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

© **Copyright International Business Machines Corporation 1988, 1993. All rights reserved.**
Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Notices xi
Trademarks and Service Marks xi

Chapter 1. Introduction 1
Who Should Read This Book 1
History of CPI Communications 1
CPI-C and the Conversational Model 1
How to Use This Book 2
General Path through the Book 3
Relationship to Products 3
Functional Levels of CPI Communications 3
CPI-C 1.0 3
CPI-C 1.1 3
X/Open Extensions to CPI-C 4
CPI-C 1.2 4
Call Table 4

Chapter 2. CPI Communications Terms and Concepts 7
Communication across an SNA Network 8
Program Partners and Conversations 9
Operating Environment 10
Side Information 11
Node Services 12
Operating System 12
Program Calls 12
Establishing a Conversation 14
Multiple Conversations 15
Contexts and Context Management 17
Conversation Characteristics 18
Modifying and Viewing Characteristics 19
Automatic Conversion of Characteristics 22
Non-Blocking Operations 22
Non-Blocking Calls 23
Outstanding Operations 23
Non-Blocking Calls and Context Management 23
Conversation Security 24
Data Conversion 24
Program Flow—States and Transitions 25
Naming Conventions—Calls and Characteristics, Variables and Values 26
Support for Resource Recovery Interfaces 28
Coordination with Resource Recovery Interfaces 28
Take-Commit and Take-Backout Notifications 29
The Backout-Required Condition 30
Responses to Take-Commit and Take-Backout Notifications 31
Additional CPI Communications States 32
Valid States for Resource Recovery Calls 33

Chapter 3. Program-to-Program Communication Tutorial 35
Interpreting the Flow Diagrams 35
Starter-Set Flows 36

© Copyright IBM Corp. 1988, 1993

Example 1: Data Flow in One Direction . . . 37
Example 2: Data Flow in Both Directions . . . 40
Advanced-Function Flows . . . 43
Data Buffering and Transmission . . . 43
Example 3: The Sending Program Changes the Data Flow Direction . . . 44
Example 4: Validation and Confirmation of Data Reception . . . 46
Example 5: The Receiving Program Changes the Data Flow Direction . . . 48
Example 6: Reporting Errors . . . 50
Example 7: Error Direction and Send-Pending State . . . 52
Example 8: Sending Program Issues a Commit . . . 54
Example 9: A Successful Commit with Conversation State Change . . . 56
Example 10: Conversation Deallocation before the Commit Call . . . 58
Example 11: Accepting Multiple Conversations Using Blocking Calls . . . 60
Example 12: Accepting Multiple Conversations Using Non-Blocking Calls . . . 62

Chapter 4. Call Reference Section . . . 65
Call Syntax . . . 66
Programming Language Considerations . . . 67
Application Generator . . . 67
C . . . 67
COBOL . . . 67
FORTRAN . . . 67
PL/I . . . 68
REXX . . . 68
RPG . . . 68
How to Use the Call References . . . 68
Locations of Key Topics . . . 69
Accept_Conversation (CMACCP) . . . 73
Accept_Incoming (CMACCI) . . . 75
Allocate (CMALLC) . . . 77
Cancel_Conversation (CMCANC) . . . 80
Confirm (CMCFM) . . . 82
Confirmed (CMCFMD) . . . 85
Convert_Incoming (CMCNVI) . . . 87
Convert_Outgoing (CMCNVO) . . . 89
Deallocate (CMDEAL) . . . 91
Extract_Conversation_Context (CMECTX) . . . 96
Extract_Conversation_State (CMECS) . . . 98
Extract_Conversation_Type (CMECT) . . . 100
Extract_Maximum_Buffer_Size (CMEMBS) . . . 102
Extract_Mode_Name (CMEMN) . . . 103
Extract_Partner_LU_Name (CMEPLN) . . . 105
Extract_Security_User_ID (CMESUI) . . . 107
Extract_Sync_Level (CMESL) . . . 109
Extract_TP_Name (CMETPN) . . . 111
Flush (CMFLUS) . . . 113
Initialize_Conversation (CMINIT) . . . 115
Initialize_For_Incoming (CMINIC) . . . 117
Prepare_To_Receive (CMPTR) . . . 119
Receive (CMRCV) . . . 123
Release_Local_TP_Name (CMRLTP) . . . 133
Request_To_Send (CMRTS) . . . 135
Send_Data (CMSEND) . . . 137
Send_Error (CMSERR) . . . 143

Set_Conversation_Security_Password (CMSCSP) . . . 149
Set_Conversation_Security_Type (CMSCST) . . . 151
Set_Conversation_Security_User_ID (CMSCSU) . . . 153
Set_Conversation_Type (CMSCT) . . . 155
Set_Deallocate_Type (CMSDT) . . . 157
Set_Error_Direction (CMSED) . . . 160
Set_Fill (CMSF) . . . 162
Set_Log_Data (CMSLD) . . . 164
Set_Mode_Name (CMSMN) . . . 166
Set_Partner_LU_Name (CMSPLN) . . . 168
Set_Prepare_To_Receive_Type (CMSPTR) . . . 170
Set_Processing_Mode (CMSPM) . . . 172
Set_Receive_Type (CMSRT) . . . 174
Set_Return_Control (CMSRC) . . . 176
Set_Send_Type (CMSST) . . . 178
Set_Sync_Level (CMSSL) . . . 180
Set_TP_Name (CMSTPN) . . . 182
Specify_Local_TP_Name (CMSLTP) . . . 184
Test_Request_To_Send_Received (CMTRTS) . . . 185
Wait_For_Conversation (CMWAIT) . . . 187

Appendix A. Variables and Characteristics . . . 191
Pseudonyms and Integer Values . . . 191
Character Sets . . . 195
Variable Types . . . 197
  Integers . . . 197
  Character Strings . . . 197

Appendix B. Return Codes . . . 201

Appendix C. State Table . . . 211
Explanation of State-Table Abbreviations . . . 211
  Conversation Characteristics ( ) . . . 212
  Return Code Values [ ] . . . 213
  data_received and status_received { , } . . . 215
  Table Symbols . . . 216
How to Use the State Table . . . 217
  Example . . . 218
Effects of Calls to the SAA Resource Recovery Interface . . . 222

Appendix D. CPI Communications and LU 6.2 . . . 225
  Send-Pending State and the error_direction Characteristic . . . 226
  Can CPI Communications Programs Communicate with APPC Programs? 227
  SNA Service Transaction Programs . . . 227
  Relationship between LU 6.2 Verbs and CPI Communications Calls . . . 227

Appendix E. Pseudonym Files . . . 231
Application Generator Pseudonym File—as Implemented by Cross System Product (CMCSP) . . . 232
C Pseudonym File (CMC or CPIC.H) . . . 250
COBOL Pseudonym File (CMCOBOL) . . . 267
FORTRAN Pseudonym File (CMFORTRN) . . . 271
PL/I Pseudonym File (CMPLI) . . . 277
REXX Pseudonym File (CMREXX) . . . 289

RPG Pseudonym File (CMRPG) 291

Appendix F. Sample Programs 297
SALESRPT (Initiator of the Conversation) 298
CREDRPT (Acceptor of the Conversation) 302
Results of Successful Program Execution 307

Summary of Changes 309

Glossary 311

Index 313

Figures

1. Programs Using CPI Communications to Converse Through an SNA Network 8
2. Operating Environment of CPI Communications Program 10
3. A Program Using Multiple Outbound CPI Communications Conversations 16
4. A Program Using Multiple Inbound CPI Communications Conversations 16
5. Server Program with Both Inbound and Outbound Conversations 18
6. Data Flow in One Direction 39
7. Data Flow in Both Directions 41
8. The Sending Program Changes the Data Flow Direction 45
9. Validation and Confirmation of Data Reception 47
10. Confirmation of Data 49
11. Reporting Errors 51
12. Error Direction and Send-Pending State 53
13. Establishing a Protected Conversation and Issuing a Successful Commit 55
14. A Successful Commit with Conversation State Change 57
15. Conversation Deallocation Precedes the Commit Call 59
16. Accepting Multiple Conversations Using Blocking Calls 61
17. Accepting Multiple Conversations Using Non-Blocking Calls 63

© Copyright IBM Corp. 1988, 1993

Tables

1. Versions of CPI Communications 4
2. Breakdown of Calls between Starter Set and Advanced Function 14
3. Characteristics and Their Default Values 20
4. Responses to Take-Commit and Take-Backout Notifications 31
5. Variables/Characteristics and Their Possible Values 192
6. Character Sets 01134 and 00640 195
7. Variable Types and Lengths 198
8. States and Transitions for CPI Communications Calls 219
9. States and Transitions for Protected Conversations 223
10. Relationship of LU 6.2 Verbs to CPI Communications Calls 228

© Copyright IBM Corp. 1988, 1993

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make them 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 that IBM program, product, or service may be used. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM authorizes any developer, under IBM's copyrights, to use and reproduce the materials in this publication ("Materials"): to implement CPI Communications in the developer's products and to describe the interface in supporting publications for those products. X/Open has authorized any developer to do the same with respect to any Materials in this publication in which X/Open holds the copyright.

The developer, in consideration of the foregoing authorization, and IBM agree not to assert any copyrights in any derivative works of the Materials against each other or any third party.

No authorization or right is granted by IBM either directly or by implication, estoppel, or otherwise other than as explicitly stated above, or under any other intellectual property right of IBM including but not limited to trade secrets, trademarks, patents, or patent applications of IBM.

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

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

Some states, provinces, and countries do not allow the exclusion of expressed or implied warranties in certain transactions; therefore, the above exclusion may not apply to you.

Trademarks and Service Marks

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:

IBM
SAA
Systems Application Architecture

© Copyright IBM Corp. 1988, 1993

Chapter 1. Introduction

This introductory chapter:

- Identifies the purpose, structure, and audience of this book
- Gives a brief overview and history of the Common Programming Interface (CPI) for Communications
- Explains how to use this book.

Who Should Read This Book

CPI Communications (CPI-C) provides a cross-system-consistent and easy-to-use programming interface for applications that require program-to-program communication. This book defines CPI Communications. It is intended for programmers who want to write applications that adhere to this definition, as well as for implementers interested in implementing CPI Communications.

Although this book is designed as an architecture specification, Chapter 3, "Program-to-Program Communication Tutorial" provides a tutorial on designing application programs using CPI Communications concepts and calls.

History of CPI Communications

CPI Communications is an evolving interface, embracing functions to meet the growing demands from different application environments and to achieve openness as an industry standard for communications programming. To date, there have been four versions of the interface:

- CPI Communications as it was first introduced in 1987, referred to in this manual as **CPI-C 1.0**
- Extensions made to CPI-C 1.0 by IBM, **CPI-C 1.1**
- Extensions made by X/Open
- CPI Communications extensions to support X/Open, **CPI-C 1.2**

This manual documents CPI-C 1.2. It is intended that CPI-C 1.2 will provide a base for the next major functional enhancement of CPI Communications.

CPI-C and the Conversational Model

The conversational model of program-to-program communications is commonly used in the industry today, and a wide variety of applications are based on this model. The model is historically thought of in terms of two applications "speaking" and "listening," hence the term "conversation." A conversation is simply a logical connection between two programs that allows the programs to communicate with each other. From an application's perspective, CPI-C provides the function necessary to enable this communication.

The conversational model is implemented in two major communications protocols, Advanced Program-to-Program Communications (APPC) and Open Systems

© Copyright IBM Corp. 1988, 1993

Interconnection Distributed Transaction Processing (OSI TP).[1] The APPC model is also referred to as logical unit type 6.2 (LU 6.2). CPI-C 1.2 provides access to both communications protocols.

A primary benefit of this design is that CPI Communications defines a single programming interface to the underlying network protocols across many different programming languages and environments. The interface's rich set of programming services shield the program from details of system connectivity and ease the integration and porting of the application programs across the supported environments.

How to Use This Book

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 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, "Pseudonym Files"**

  This appendix contains sample CPI Communications pseudonym files for several programming languages.

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

  This appendix contains two sample COBOL programs using CPI Communications.

[1] International Standardization Organization (ISO) and International Electrotechnical Commission (IEC) standard 10026-1: 1992, *Information Technology — Open Systems Interconnection — Distributed Transaction Processing*.

General Path through the Book

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 an 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

CPI Communications defines a programming interface that is available across a wide variety of systems. Preparing and running programs requires the use of a CPI Communications product that implements the interface on one of these systems. These products have their own sets of product documentation, which will be required in addition to this book.

Functional Levels of CPI Communications

CPI-C 1.0

The initial specification of CPI Communications provided a standard base for conversational communications:

- Ability to start and end conversations
- Support for program synchronization through confirmation flows
- Error processing
- Ability to optimize conversation flow (using Flush and Prepare_To_Receive calls).

CPI-C 1.1

CPI-C 1.0 was extended in 1990 to include four areas of new function:

- Support for resource recovery
- Automatic parameter conversion
- Support for communication with non-CPI-C programs
- Local/remote transparency.

Note: For more information about the CPI-C 1.1 architecture, see the seventh edition of *SAA Common Programming Interface: Communications Reference*, SC26-4399-06.

X/Open Extensions to CPI-C

X/Open adopted CPI-C at the 1.1 level (with the exception of support for resource recovery) to allow X/Open-compliant systems to communicate with systems implementing LU 6.2. The developers specification published in 1990 included several new functions not found in CPI-C 1.1:

- Support for non-blocking calls
- Ability to accept multiple conversations
- Support for data conversion (beyond parameters)
- Support for security parameters.

CPI-C 1.2

CPI-C 1.0 was designed to provide a consistent programming interface for communications programming. Unfortunately, each of its derivatives, namely CPI-C 1.1 and X/Open CPI-C, provided different levels of function. CPI-C 1.2 (defined in this manual) consolidates CPI-C 1.1 and the X/OPEN extensions, providing function in four areas:

- Support for non-blocking calls—incorporation of X/Open calls
- Support for data conversion—incorporation of X/Open calls
- Support for specification of security parameters—incorporation of X/Open calls
- Ability to accept multiple conversations—new calls to accommodate both the X/Open and CPI-C 1.1 approaches.

Call Table

Table 1 lists the calls defined for the different versions of CPI Communications. An X is used to indicate that the call was part of a specific version.

Table 1 (Page 1 of 2). Versions of CPI Communications

Call Name	CPIC 1.0	CPIC 1.1	CPIC 1.2
Starter Set			
Accept_Conversation	X	X	X
Allocate	X	X	X
Deallocate	X	X	X
Initialize_Conversation	X	X	X
Receive	X	X	X
Send_Data	X	X	X
Advanced Function			
for synchronization and control:			
Confirm	X	X	X
Confirmed	X	X	X
Flush	X	X	X
Prepare_To_Receive	X	X	X
Request_To_Send	X	X	X
Send_Error	X	X	X
Test_Request_To_Send_Received	X	X	X
for modifying conversation characteristics:			
Set_Conversation_Type	X	X	X
Set_Deallocate_Type	X	X	X
Set_Error_Direction	X	X	X

Table 1 (Page 2 of 2). Versions of CPI Communications

Call Name	CPIC 1.0	CPIC 1.1	CPIC 1.2
Set_Fill	X	X	X
Set_Log_Data	X	X	X
Set_Mode_Name	X	X	X
Set_Partner_LU_Name	X	X	X
Set_Prepare_To_Receive_Type	X	X	X
Set_Receive_Type	X	X	X
Set_Return_Control	X	X	X
Set_Send_Type	X	X	X
Set_Sync_Level			
CM_NONE	X	X	X
CM_CONFIRM	X	X	X
CM_SYNC_POINT		X	X
Set_TP_Name	X	X	X
for examining conversation characteristics:			
Extract_Conversation_Context			X
Extract_Conversation_State		X	X
Extract_Conversation_Type	X	X	X
Extract_Mode_Name	X	X	X
Extract_Partner_LU_Name	X	X	X
Extract_Security_User_ID			X
Extract_Sync_Level	X	X	X
Extract_TP_Name			X
for non-blocking operations:			
Cancel_Conversation			X
Set_Processing_Mode			X
Wait_For_Conversation			X
for accepting multiple conversations:			
Accept_Incoming			X
Initialize_For_Incoming			X
Release_Local_TP_Name			X
Specify_Local_TP_Name			X
for security:			
Set_Conversation_Security_Password			X
Set_Conversation_Security_Type			X
Set_Conversation_Security_User_ID			X
for data conversion:			
Convert_Incoming			X
Convert_Outgoing			X
for determining buffer size:			
Extract_Maximum_Buffer_Size			X

Introduction

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 provides access to 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.

© Copyright IBM Corp. 1988, 1993

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.

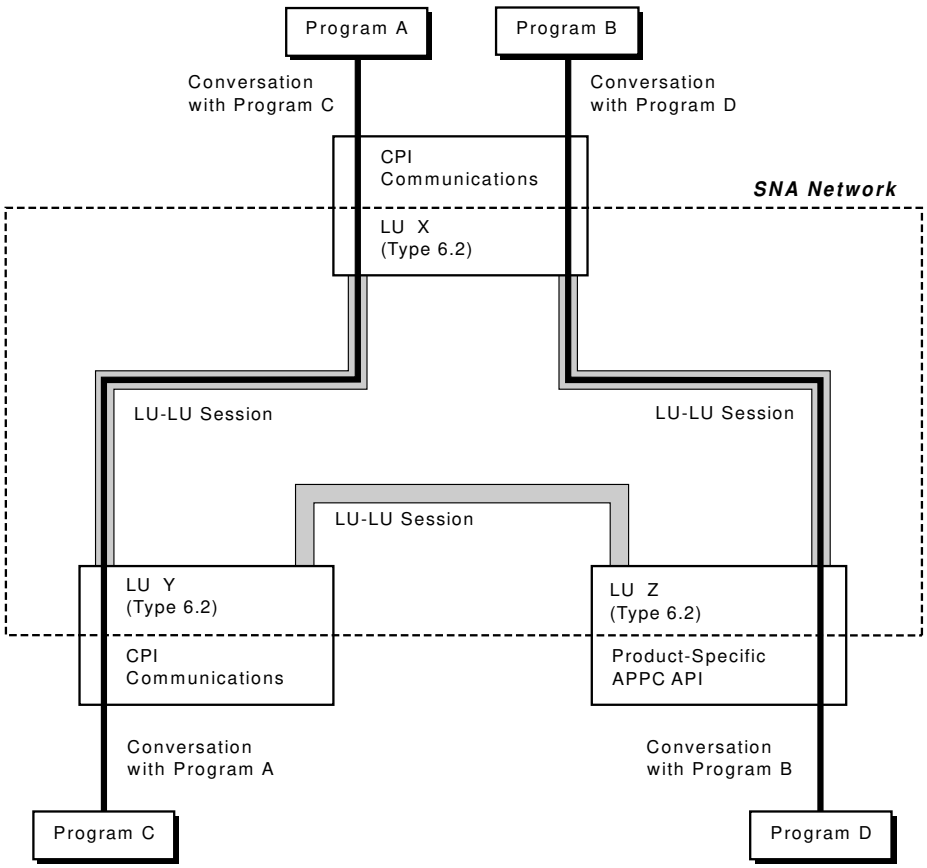


Figure 1. Programs Using CPI Communications to Converse Through an SNA Network

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* (GC30-3072) and *Systems Network Architecture Technical Overview* (GC30-3073).

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 137.

For further information on basic and mapped conversations, refer to *SNA LU 6.2 Reference: Peer Protocols* (SC31-6808) and *SNA Transaction Programmer's Reference Manual for LU Type 6.2* (GC30-3084).

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 circumstances.

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.

Note: A CPI Communications program may establish a conversation with a program that is using the LU 6.2 application programming interface (API) 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 225 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 8. The new line on the left shows Program A's communication with the communications element itself. This line represents a specific set of program calls that can be made using the local system's library of common code. The different types of CPI Communications calls that a program may make are discussed later in this chapter in "Program Calls."

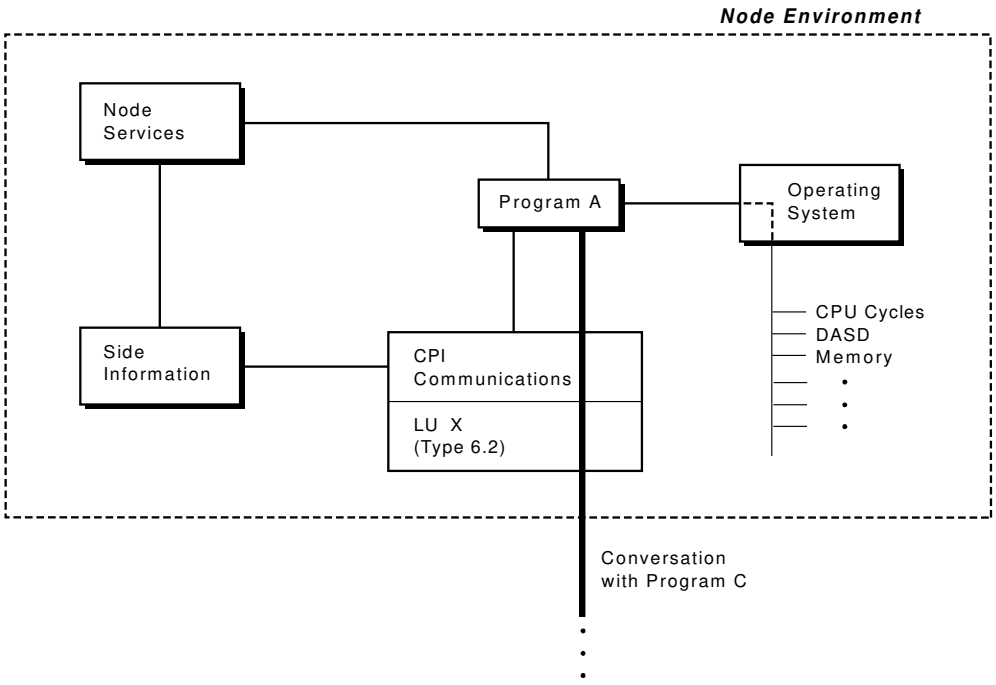


Figure 2. Operating Environment of CPI Communications Program

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

- Side information
- Node services
- Operating system

These new elements are discussed in the following sections.

Side Information

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

System administrators supply and maintain the side information for 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 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 225 for details of how a CPI Communications program can interact with non-CPI Communications programs.

- ***conversation_security_type***

  Specifies the type of access security information to be included in the conversation startup request. See "Conversation Security" on page 24 for more information.

- ***security_user_ID***

  Specifies the user ID to be used for validation of access to the remote program by the partner LU.

- ***security_password***

  Specifies the password to be used with the user ID for validation of access to the remote program by the partner LU.

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 115.

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 36) or by local (operator) action. In either case, node services sets up the required data paths and operating environment required by the program, validates and establishes security parameters under which the program will execute, 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 any access security information included in the conversation startup request, 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.

- Support for programs with multiple partners

  As is discussed in greater detail in "Multiple Conversations" on page 15, some programs do work on behalf of multiple partners. Each partner is represented by a **context**, or collection of logical attributes. Node Services provides the necessary function and support (through system interfaces) to allow the program to manage different partner contexts. See "Contexts and Context Management" on page 17 for more information.

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. An example of a conversation characteristic is the *conversation_type* characteristic, which indicates whether the conversation is basic or mapped. Conversation characteristics are discussed in greater detail in "Conversation Characteristics" on page 18.

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 36.

- **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 43.

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

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

Starter Set	
Initialize_Conversation	
Accept_Conversation	
Allocate	
Send_Data	
Receive	
Deallocate	
Advanced Function	
Accept_Incoming	Set_Conversation_Security_Password
Cancel_Conversation	Set_Conversation_Security_Type
Confirm	Set_Conversation_Security_User_ID
Confirmed	Set_Conversation_Type
Convert_Incoming	Set_Deallocate_Type
Convert_Outgoing	Set_Error_Direction
Flush	Set_Fill
Initialize_For_Incoming	Set_Log_Data
Prepare_To_Receive	Set_Mode_Name
Release_Local_TP_Name	Set_Partner_LU_Name
Request_To_Send	Set_Prepare_To_Receive_Type
Send_Error	Set_Processing_Mode
Specify_Local_TP_Name	Set_Receive_Type
Test_Request_To_Send_Received	Set_Return_Control
Wait_For_Conversation	Set_Send_Type
	Set_Sync_Level
Extract_Conversation_Context	Set_TP_Name
Extract_Security_User_ID	
Extract_Conversation_State	
Extract_Conversation_Type	
Extract_Maximum_Buffer_Size	
Extract_Mode_Name	
Extract_Partner_LU_Name	
Extract_Sync_Level	
Extract_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 70.

Establishing a 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. The conversation startup request contains information necessary to start the partner program and establish the conversation.

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). Program C will use its *conversation_ID* in all future calls intended for that conversation.

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. The default values established by CPI Communications are discussed in "Conversation Characteristics" on page 18.

Multiple Conversations

In the example above, program A established a single conversation with a single partner, but CPI Communications allows a program to communicate with multiple partners using multiple, concurrent conversations:

- **Outbound Conversations**—A program initiates more than one conversation.
- **Inbound Conversations**—A program accepts more than one conversation.

Specific combinations of outbound and inbound conversations are determined by application design. The sections that follow discuss in greater detail the concepts required for multiple conversations.

Naming of Partner Programs

After a program issues Initialize_Conversation to establish its conversation characteristics, a name for its partner program (the TP_Name) is established. This name is transmitted to the remote system in the conversation startup request after the program issues the Allocate call.

At the remote system, the partner program can be started in one of two ways:

- Receipt of a conversation startup request
- Local action.

In the first case, node services starts the program named in the conversation startup request. However, if a program is started locally, the program must notify node services of its ability to accept conversations for a given name. The program **associates** a name with itself by issuing a Specify_Local_TP_Name call. The program can **release** a name from association with itself by issuing the Release_Local_TP_Name call.

To accept multiple conversations for different names, the program issues multiple Specify_Local_TP_Name calls, thus associating multiple names with itself.

Note: A locally-started program cannot accept conversations until a name has been associated with the program.

Multiple Outbound Conversations

Figure 3 on page 16 shows program A establishing conversations with two partners. For example, a program may need to request data from multiple data bases on different nodes to answer a particular query. The conversation with Program B is initialized with an Initialize_Conversation (CMINIT) call that returns a *conversation_ID* parameter of X. The conversation with Program C is initialized with an Initialize_Conversation call that returns a *conversation_ID* parameter 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.

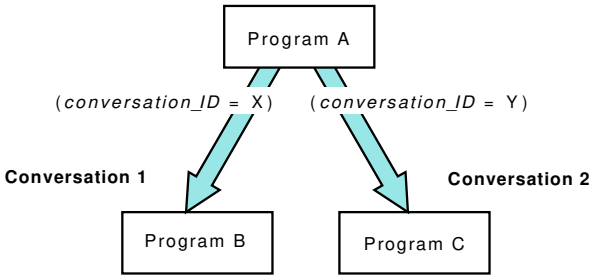


Figure 3. A Program Using Multiple Outbound CPI Communications Conversations

Note: In some implementing environments, Program A can share the *conversation_ID* with another task, allowing that task to issue calls on the conversation with Program C.

Multiple Inbound Conversations

Some programs, often referred to as **server** programs, may need to accept more than one inbound conversation. For example, a server could accept conversations from multiple partners in order to work on the request from one partner while waiting for a second partner's request or work to complete.

This type of application is shown in Figure 4, where Programs D and E have both chosen to initiate conversations with the same partner, Program S.

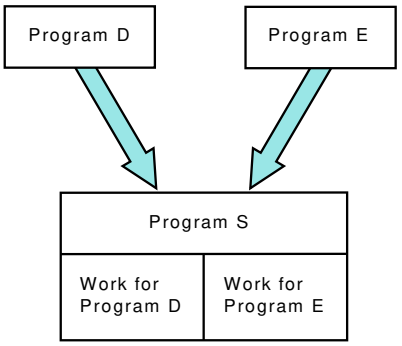


Figure 4. A Program Using Multiple Inbound CPI Communications Conversations

In the simplest case, Program S can accept the two conversations by issuing Accept_Conversation twice. Alternatively, Program S may make use of two advanced calls, Initialize_For_Incoming and Accept_Incoming.

Note: A program would use the advanced calls to achieve greater programming flexibility. See “Non-Blocking Operations” on page 22 for a more detailed discussion. See “Example 11: Accepting Multiple Conversations Using Blocking Calls” on page 60 and “Example 12: Accepting Multiple Conversations Using Non-Blocking Calls” on page 62 for examples using these calls.

Contexts and Context Management

Node services provides support for programs that perform work on behalf of multiple partners, such as server Program S in the example above. Each time a program accepts an incoming conversation, a new **context** is created. The context is identified by a system-wide **context identifier** and is used by node services to group logical attributes for the work to be done on behalf of the partner program.

Node services maintains one or more contexts for a program in execution within the node. For each program, there is one distinguished context, the **current context**, within which work is currently being done. A program can manage different contexts by making calls to node services in order to:

- Create a new context
- Terminate a context
- Set the current context
- Retrieve the context identifier of the current context.

Note: The discussion of contexts throughout this and following sections assumes a context is maintained on a program basis. However, in a system that supports multi-threaded programs, the context may be maintained on a thread basis.

Relationship between Contexts and Security Parameters

Security parameters are among the logical attributes maintained by node services for a context. The access security information carried in the conversation startup request is used to set the security parameters for the context created as a result of an incoming conversation. The security parameters for other contexts are based on security information maintained within the system. See “Conversation Security” on page 24 for a complete discussion of conversation security.

Relationship between Conversations and Contexts

Each conversation is assigned to a context when it is allocated or accepted.

- An outgoing conversation is assigned to the current context of the program when it issues the Allocate call.
- An incoming conversation is assigned to the new context created when the program accepts the incoming conversation with the Accept_Conversation or Accept_Incoming call.

A program can retrieve the context identifier for a conversation's context by issuing the Extract_Conversation_Context call.

The program's current context is set by node services to the newly created context when an Accept_Conversation or Accept_Incoming call completes successfully with *return_code* set to CM_OK.

Inbound and Outbound Conversations

A program that accepts incoming conversations from multiple program partners must ensure that work is done within the right context. In the expanded server example shown in Figure 5, Program S accepts two conversations, one each from Programs D and E. Two new contexts are created, one for the work done on behalf of Program D and one for the work done on behalf of Program E. Program S can retrieve the context identifier for each conversation by issuing the Extract_Conversation_Context call twice.

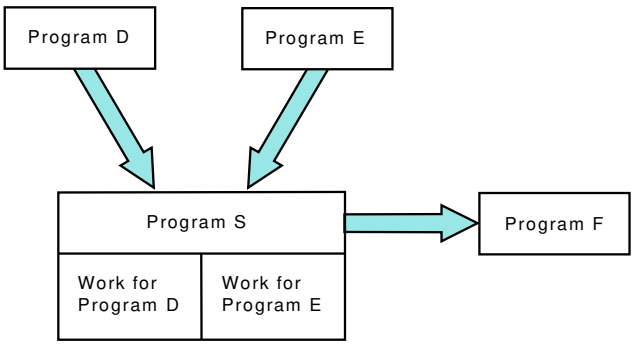


Figure 5. Server Program with Both Inbound and Outbound Conversations

Now consider a scenario where Program S is doing work for Program E and receives a request from Program D. The request is for information that Program S does not have. In this case, Program S allocates a conversation to a third partner, Program F, in order to answer Program D's request.

Before allocating the conversation to Program F, Program S must first ensure that it is using the correct context. It does this by setting the current context to that for Program D. This causes the outgoing conversation to be established using information, such as the proper security parameters, from the correct context. “Conversation Security” on page 24 provides a complete discussion of how the current context is used by node services to establish security parameters for new conversations.

For more information about establishing and managing conversations, see the following sections:

- “Non-Blocking Operations” on page 22 for use of non-blocking calls
- “Example 1: Data Flow in One Direction” on page 37
- “Example 11: Accepting Multiple Conversations Using Blocking Calls” on page 60
- “Example 12: Accepting Multiple Conversations Using Non-Blocking Calls” on page 62
- Usage notes for “Accept_Conversation (CMACCP)” on page 73
- Usage notes for “Accept_Incoming (CMACCI)” on page 75
- Usage notes for “Initialize_Conversation (CMINIT)” on page 115
- Usage notes for “Initialize_For_Incoming (CMINIC)” on page 117.

Conversation Characteristics

As discussed above, 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, and the initial values assigned to the characteristics depend on the program's role in starting the conversation. Table 3 on page 20 provides a comparison of the conversation characteristics and initial values as set by the Initialize_Conversation, Accept_Conversation, Initialize_For_Incoming, and Accept_Incoming calls. The uppercase values shown in the table 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 26.

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.”

Table 3 (Page 1 of 2). Characteristics and Their Default Values

Name of Characteristic	Initialize_Conversation sets it to:	Accept_Conversation sets it to:	Initialize_For_Incoming sets it to:	Accept_Incoming sets it to:
context_ID	Not set	The *context_ID* of the newly created context	Not set	The *context_ID* of the newly created context
context_ID_length	Not set	The length of *context_ID*	Not set	The length of *context_ID*
conversation_security_type	The security type from side information referenced by *sym_dest_name*. If a blank *sym_dest_name* was specified, *conversation_security_type* will be CM_SECURITY_SAME.	Not applicable	Not applicable	Not applicable
conversation_state	CM_INITIALIZE_STATE	CM_RECEIVE_STATE	CM_INITIALIZE_INCOMING_STATE	CM_RECEIVE_STATE
conversation_type	CM_MAPPED_CONVERSATION	The value received on the conversation startup request	Not set	The value received on the conversation startup request
deallocate_type	CM_DEALLOCATE_SYNC_LEVEL	CM_DEALLOCATE_SYNC_LEVEL	CM_DEALLOCATE_SYNC_LEVEL	Not changed by Accept_Incoming
error_direction	CM_RECEIVE_ERROR	CM_RECEIVE_ERROR	CM_RECEIVE_ERROR	Not changed by Accept_Incoming
fill	CM_FILL_LL	CM_FILL_LL	CM_FILL_LL	Not changed by Accept_Incoming
log_data	Null	Null	Null	Not changed by Accept_Incoming
log_data_length	0	0	0	Not changed by Accept_Incoming
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	Not set	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*	Not set	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	Not set	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*	Not set	The length of *partner_LU_name*
prepare_to_receive_type	CM_PREP_TO_RECEIVE_SYNC_LEVEL	CM_PREP_TO_RECEIVE_SYNC_LEVEL	CM_PREP_TO_RECEIVE_SYNC_LEVEL	Not changed by Accept_Incoming
processing_mode	CM_BLOCKING	CM_BLOCKING	CM_BLOCKING	Not changed by Accept_Incoming
receive_type	CM_RECEIVE_AND_WAIT	CM_RECEIVE_AND_WAIT	CM_RECEIVE_AND_WAIT	Not changed by Accept_Incoming
return_control	CM_WHEN_SESSION_ALLOCATED	Not applicable	Not applicable	Not applicable

Table 3 (Page 2 of 2). Characteristics and Their Default Values

Name of Characteristic	Initialize_Conversation sets it to:	Accept_Conversation sets it to:	Initialize_For_Incoming sets it to:	Accept_Incoming sets it to:
security_password	The security password from side information referenced by *sym_dest_name*. If a blank *sym_dest_name* was specified, *security_password* will be the null string.	Not applicable	Not applicable	Not applicable
security_password_length	The length of *security_password*. If a blank *sym_dest_name* was specified, *security_password_length* will be 0.	Not applicable	Not applicable	Not applicable
security_user_ID	The security user ID from side information referenced by *sym_dest_name*. If a blank *sym_dest_name* was specified, *security_user_ID* will be the null string.	The value received on the conversation startup request	Not set	The value received on the conversation startup request
security_user_ID_length	The length of *security_user_ID*. If a blank *sym_dest_name* was specified, *security_user_ID_length* will be 0.	The length of *security_user_ID*	Not set	The length of *security_user_ID*
send_type	CM_BUFFER_DATA	CM_BUFFER_DATA	CM_BUFFER_DATA	Not changed by Accept_Incoming
sync_level	CM_NONE	The value received on the conversation startup request	Not set	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.	The value received on the conversation startup request	Not set	The value received on the conversation startup request
TP_name_length	The length of *TP_name*. If a blank *sym_dest_name* was specified, *TP_name_length* will be 1.	The length of *TP_name*	Not set	The length of *TP_name*

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:

security_password	Specified on the Set_Conversation_Security_Password call.
security_user_ID	Specified on the Extract_Security_User_ID and Set_Conversation_Security_User_ID calls.
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.

Non-Blocking Operations

CPI Communications supports two processing modes for its calls:

- **Blocking**—The call operation completes before control is returned to the program. If the call operation is unable to complete immediately, it “blocks,” and the program is forced to wait until the call operation finishes. While waiting, the program is unable to perform other processing or to communicate with any of its other partners.
- **Non-blocking**—If possible, the call operation completes immediately and control is returned to the program. However, if while processing the call CPI Communications determines that the call operation cannot complete immediately, control is returned to the program even though the call operation has not completed. The call operation remains in progress, and completion of the call operation occurs at a later time.

The processing mode for a call is determined by the *processing_mode* conversation characteristic, which has a default value of CM_BLOCKING. The program chooses non-blocking operations by setting the *processing_mode* characteristic to CM_NON_BLOCKING.

The sections that follow provide a detailed discussion of the concepts required for use of non-blocking calls.

Non-Blocking Calls

When *processing_mode* is set to CM_NON_BLOCKING for a conversation and a call operation cannot complete immediately, CPI Communications returns control to the program with a *return_code* of CM_OPERATION_INCOMPLETE. Thus, the program is allowed to perform other processing while the previous call operation remains in progress. The following calls can return the CM_OPERATION_INCOMPLETE return code:

- Accept_Incoming
- Allocate
- Confirm
- Confirmed
- Deallocate
- Flush
- Prepare_To_Receive
- Receive
- Request_To_Send
- Send_Data
- Send_Error

Outstanding Operations

When a call receives the CM_OPERATION_INCOMPLETE return code, the operation remains in progress as an **outstanding operation**. The program must issue the Wait_For_Conversation call to determine when an outstanding operation is completed and to retrieve the return code for that operation. CPI Communications keeps track of all conversations with outstanding operations and responds to a subsequent Wait_For_Conversation call with the conversation identifier of one of those conversations when the operation on it completes.

CPI Communications allows one outstanding operation per conversation. The only valid calls on a conversation with an outstanding operation are Wait_For_Conversation and Cancel_Conversation. Any other call on the conversation will return the CM_OPERATION_NOT_ACCEPTED return code.

A conversation does not change conversation state when a call on that conversation gets the CM_OPERATION_INCOMPLETE return code. Instead, the state transition occurs when a subsequent Wait_For_Conversation call completes and indicates that the conversation has a completed operation. The conversation enters the state called for by a combination of the operation that completed, the return code for that operation (the *conversation_return_code* value returned on the Wait_For_Conversation call), and the other factors that determine state transitions.

A program may use the Cancel_Conversation call to end a conversation and any outstanding operation on that conversation.

Non-Blocking Calls and Context Management

In general, the program's current context is set by node services to the newly created context when an Accept_Conversation or Accept_Incoming call completes successfully with a return code of CM_OK. However, if Accept_Incoming is issued with *processing_mode* set to CM_NON_BLOCKING and returns with a CM_OPERATION_INCOMPLETE return code, the program's current context will not be changed. A new context is not created until the Accept_Incoming call operation subsequently completes successfully as a result of the Wait_for_Conversation call.

The program can then use the Extract_Conversation_Context call to determine the context assigned to the conversation. Because Wait_for_Conversation does not cause a change of context, the program is responsible for issuing the appropriate node services call to establish the correct current context.

Conversation Security

Many systems control access to system resources through security parameters associated with a request for access to those resources. In particular, an LU working in conjunction with node services can control access to its programs and conversation resources using access security information carried in the conversation startup request.

Access security information may contain neither, one (only the user ID), or both of the following two pieces of information:

- User ID of the user on whose behalf access to the remote program is requested
- Password to be used with the user ID to validate access to the remote program.

The access security information in the conversation startup request depends on the values of the security conversation characteristics and comes from the following sources:

- The system administrator can provide security parameters in the side information. These are used to establish security characteristics when the program issues the Initialize_Conversation call.
- The program can override the values from side information and set the security characteristics directly using the Set_Conversation_Security_Type, Set_Conversation_Security_User_ID, and Set_Conversation_Security_Password calls.
- Node services maintains security parameters for the program's contexts. Security parameters for the program's current context supply the access security information in the conversation startup request when the program allocates a conversation with *conversation_security_type* set to CM_SECURITY_SAME.

When a program is started as a result of an incoming conversation startup request or when an already started program accepts an incoming conversation, node services uses the access security information to validate the user's access to the program and to establish the security parameters for the resulting context.

The program that accepts an incoming conversation may examine the *security_user_ID* for that conversation by issuing the Extract_Security_User_ID call.

Data Conversion

Program-to-program communication typically involves a variety of computer systems and languages. Because each system or language has its own way of representing equivalent data, data conversion support is needed for the application program to overcome the differences in data representations from different environments.

With the Convert_Incoming and Convert_Outgoing calls, CPI Communications provides data conversion for character data in the user buffer. These calls may be used to write a program that is independent of the encoding of the partner program:

- Before issuing a Send_Data call, the program may issue the Convert_Outgoing call to convert the application data in the local encoding to the corresponding EBCDIC hexadecimal codes.
- After receiving data from a **Receive** call, the program may issue the Convert_Incoming call to convert the EBCDIC hexadecimal codes to the corresponding local representation of the data.

These two calls provide limited data conversion support for character data that belongs to character set 00640, as specified in Appendix A, "Variables and Characteristics." See the usage notes under "Convert_Incoming (CMCNVI)" on page 87 and "Convert_Outgoing (CMCNVO)" on page 89.

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 52 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.

State	Description
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.
Initialize-Incoming	Initialize_For_Incoming has completed successfully and a *conversation_ID* has been assigned for this conversation. The program may accept an incoming conversation by issuing Accept_Incoming on this conversation.

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 Resource Recovery Interfaces" on page 28 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.

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 192 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 198.
- Table 5 on page 192 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 67 in Chapter 4, “Call Reference Section” provides information on system files that can be used to establish pseudonyms for a program.

Support for Resource Recovery Interfaces

This section describes how application programs can use CPI Communications in conjunction with a resource recovery interface. A **resource recovery interface** provides access to services and facilities that use two-phase commit protocols to coordinate changes to distributed resources.

While CPI Communications' sync point functions can be used with other resource recovery interfaces, this book uses the Systems Application Architecture* (SAA*) resource recovery interface in its examples that illustrate how CPI Communications works with resource recovery interfaces. For information about performing synchronization point processing with the SAA resource recovery interface, see *SAA Common Programming Interface: Resource Recovery Reference* (SC31-6821) and read the documentation for the appropriate operating environment.

Note: 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.

A CPI Communications conversation can be used with a 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 Resource Recovery Interfaces

A program communicates with a 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 a resource recovery interface's commit call to establish a new sync point or a resource recovery interface's 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.

The commitment or backout of protected resources is done on a context basis. Only those changes to protected resources, including protected conversations, that belong to the program's current context when the commit or backout call is issued are committed or backed out. Therefore, prior to issuing the commit or backout call, the program must ensure that the current context is the context for which the commit or backout is intended.

For example, Program S in Figure 4 on page 16 has two distinct contexts, one for Program D and one for Program E. If Program S decides to commit or back out the work that it has done for Program D, it must set the current context to ensure

that only the resources associated with Program D will be affected. For more information on contexts, see “Contexts and Context Management” on page 17.

Take-Commit and Take-Backout Notifications

When a program issues a commit or backout call, CPI Communications cooperates with the 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 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 a 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 a 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 a 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 (basic conversations only)

- CM_DEALLOCATED_ABEND_TIMER_BO (basic conversations only)
- 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 a protected conversation, the conversation's context is placed in the **Backout-Required** condition. This condition is not a conversation state, because it applies to all of the program's protected resources for that context, possibly including multiple conversations. When a context is in the **Backout-Required** condition, there are restrictions on the CPI Communications calls that may be used on protected conversations within that context.

A context 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 context in the **Backout-Required** condition.

When a context is placed in the **Backout-Required** condition, the program should issue a resource recovery 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 protected conversations within that context. 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 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 conversation's context is placed in the **Backout-Required** condition.	The program should issue a resource recovery 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 conversation's context is placed in the **Backout-Required** condition.	The program should issue a resource recovery 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 25, 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 a resource recovery commit call or a CPI Communications Flush or Confirm call. A conversation enters **Defer-Receive** state when the local program issues 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, or when 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 Resource Recovery Calls

A program must ensure that there are no outstanding operations on its protected conversations within a context before issuing a resource recovery call for that context. If a resource recovery call is issued while there is an outstanding operation on a protected conversation, the program receives a program-state-check return code from the resource recovery interface.

All protected conversations of a program must be in one of the following states for the program to issue a 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 a program-state-check return code from the resource recovery interface. 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.

A backout call can be issued in any state.

Note: When the SAA resource recovery interface is used, the program-state-check return code is RR_PROGRAM_STATE_CHECK.

Terms and Concepts

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 36
- "Advanced-Function Flows" on page 43.

In addition to these sample flows, a simple COBOL application using CPI Communications calls is provided in Appendix F, "Sample Programs" on page 297.

Interpreting the Flow Diagrams

In the flow diagrams shown in this chapter (for example, Figure 6 on page 39), 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."

Note: Except for "Example 12: Accepting Multiple Conversations Using Non-Blocking Calls," these example flows assume that the *processing_mode* characteristic is set to the default value of CM_BLOCKING.

© Copyright IBM Corp. 1988, 1993

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 37 demonstrates a flow of data in only one direction (only the initiating program sends data).
- “Example 2: Data Flow in Both Directions” on page 40 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 6 on page 39 shows an example of a conversation where the flow of data is in only one direction.

The steps shown in Figure 6 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 115, 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 77) using the *conversation_ID* previously assigned by the Initialize_Conversation call.
4	If a session between the systems 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 system 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 77 for more information about the error conditions that can result.

Step	Description
6 and 7	Program A sends data with the Send_Data call (described in "Send_Data (CMSEND)" on page 137) 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 system'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. **Notes:** 1. Some implementations may choose to transmit the conversation startup request as part of the Allocate processing. 2. The local program can ensure that the remote program is connected as soon as possible by issuing Flush (CMFLUS) immediately after Allocate (CMALLC). 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 43.
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 73 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 123). 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 91) to send any data buffered in the local system and release the conversation. Program C issues a final Receive, shown here in the same step as the Deallocate, to check that it has all the received data.
13 and 14	The *return_code* of CM_DEALLOCATED_NORMAL tells Program C that the conversation is deallocated. Both Program C and Program A finish normally. **Note:** Only one program should issue Deallocate; in this case it was Program A. If Program C had issued Deallocate after receiving CM_DEALLOCATED_NORMAL, an error would have resulted.

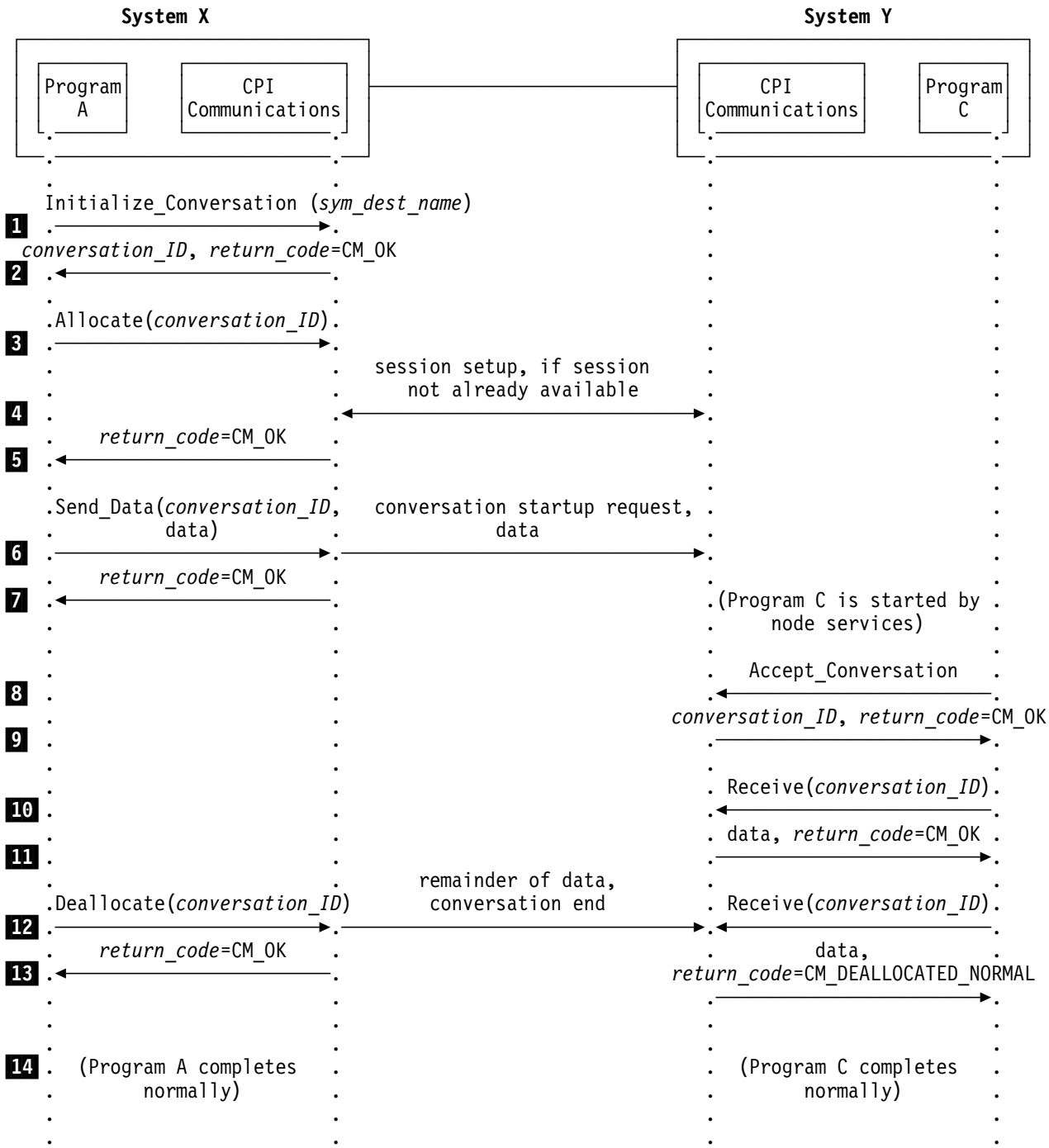


Figure 6. Data Flow in One Direction

Example 2: Data Flow in Both Directions

Figure 7 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 7 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 123, 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 44 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.

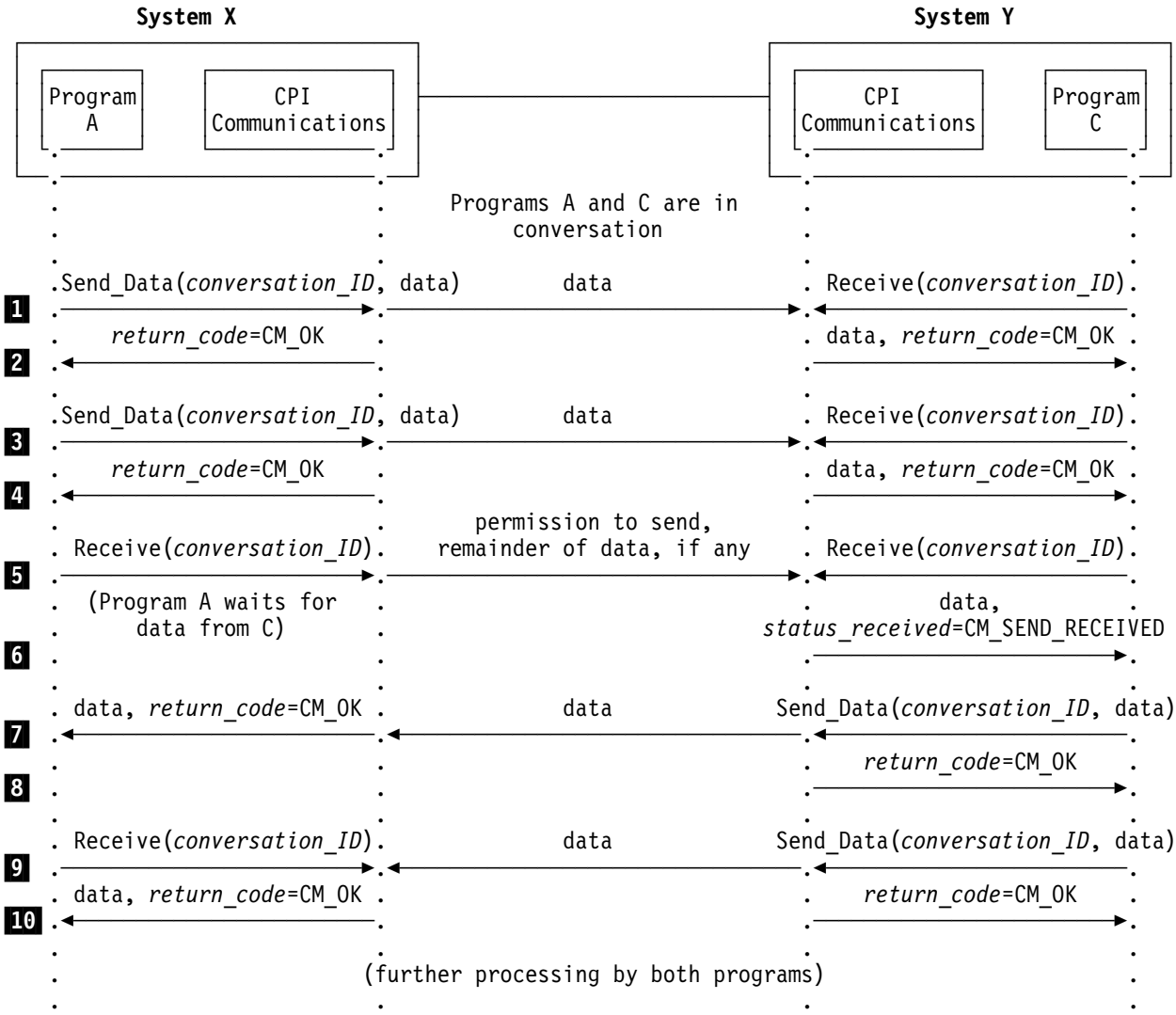


Figure 7. Data Flow in Both Directions

Tutorial

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 a resource recovery commit call to establish a synchronization point for protected resources.
- "Example 9: A Successful Commit with Conversation State Change" shows a successful resource recovery commit operation with a conversation state change.
- "Example 10: Conversation Deallocation before the Commit Call" shows a resource recovery commit call issued after a CPI Communications Deallocate call.
- "Example 11: Accepting Multiple Conversations Using Blocking Calls" shows a program that uses blocking calls to accept multiple incoming conversations.
- "Example 12: Accepting Multiple Conversations Using Non-Blocking Calls" shows a program that uses non-blocking calls to accept multiple incoming conversations.

This section begins with a discussion of how a program can exercise control over when the system 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 system 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 system 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 36. The other calls are discussed in the following examples.

Example 3: The Sending Program Changes the Data Flow Direction

Figure 8 is a variation on the function provided by the flow shown in "Example 2: Data Flow in Both Directions" on page 40. When the data flow direction changes, Program A can continue processing instead of waiting for data to arrive.

The steps shown in Figure 8 are:

Step	Description
1 through **6**	The program begins the same as "Example 1: Data Flow in One Direction" on page 37. 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 119), 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 8. For further discussion of this alternate flow, see "Set_Receive_Type (CMSRT)" on page 174 and "Receive (CMRCV)" on page 123.

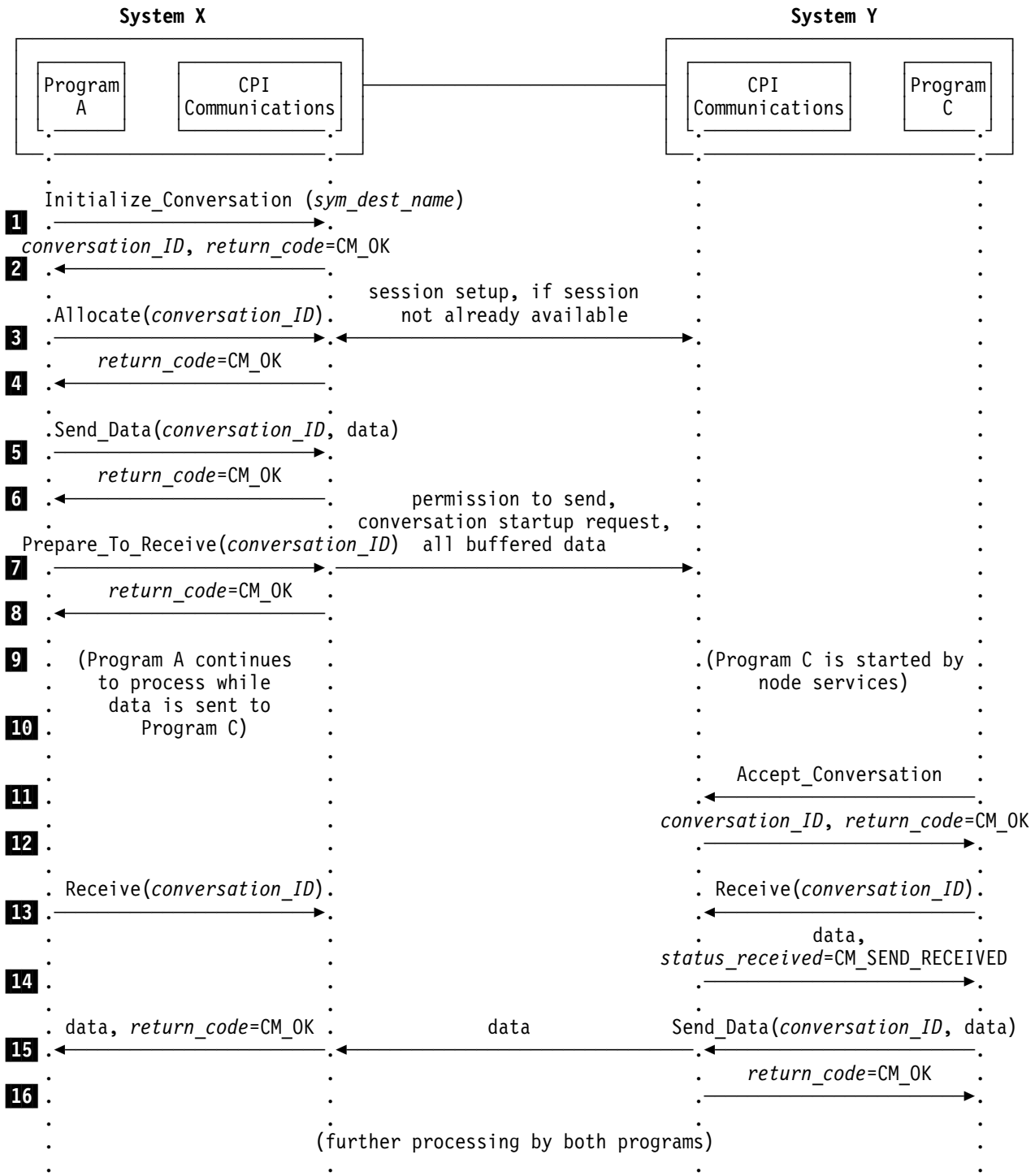


Figure 8. The Sending Program Changes the Data Flow Direction

Example 4: Validation and Confirmation of Data Reception

Figure 9 on page 47 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 9 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 180 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 113) to make sure that the conversation is immediately established. If data is present, the local system buffer is emptied and the contents are sent to the remote system. 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 performed any data validation that Programs A and C have agreed upon. Program A is forced to wait for a reply.
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 85), the conversation returns to **Receive** state. Meanwhile, at System X, the confirmation reply arrives and the CM_OK *return_code* is sent back to Program A.
16	Program A continues with further processing. **Note:** Unlike the previous examples in which a program could bypass waiting, this example demonstrates that use of the Confirm call forces the program to wait for a reply.

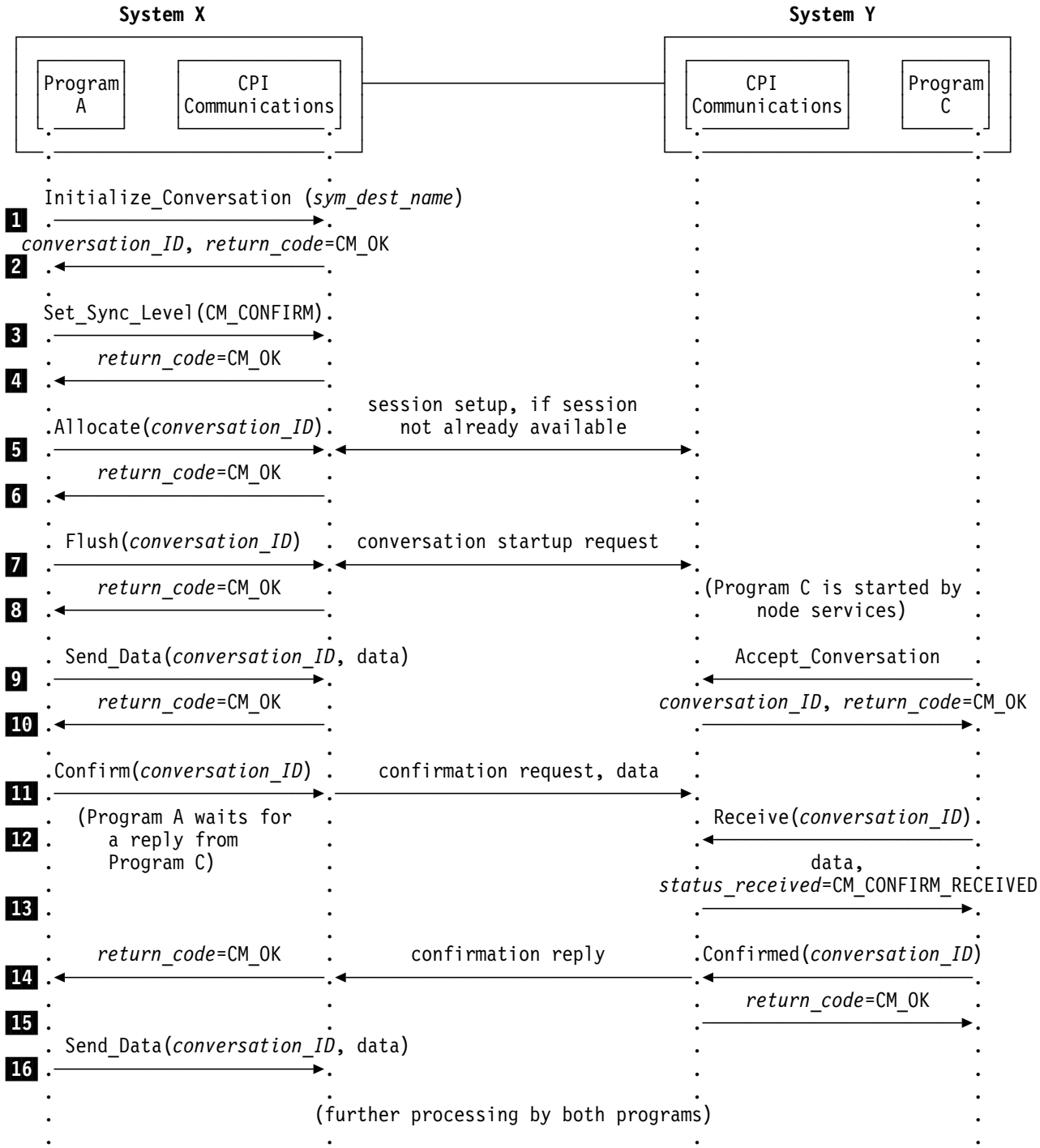


Figure 9. Validation and Confirmation of Data Reception

Example 5: The Receiving Program Changes the Data Flow Direction

Figure 10 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 135 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 10 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 44 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 system'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.

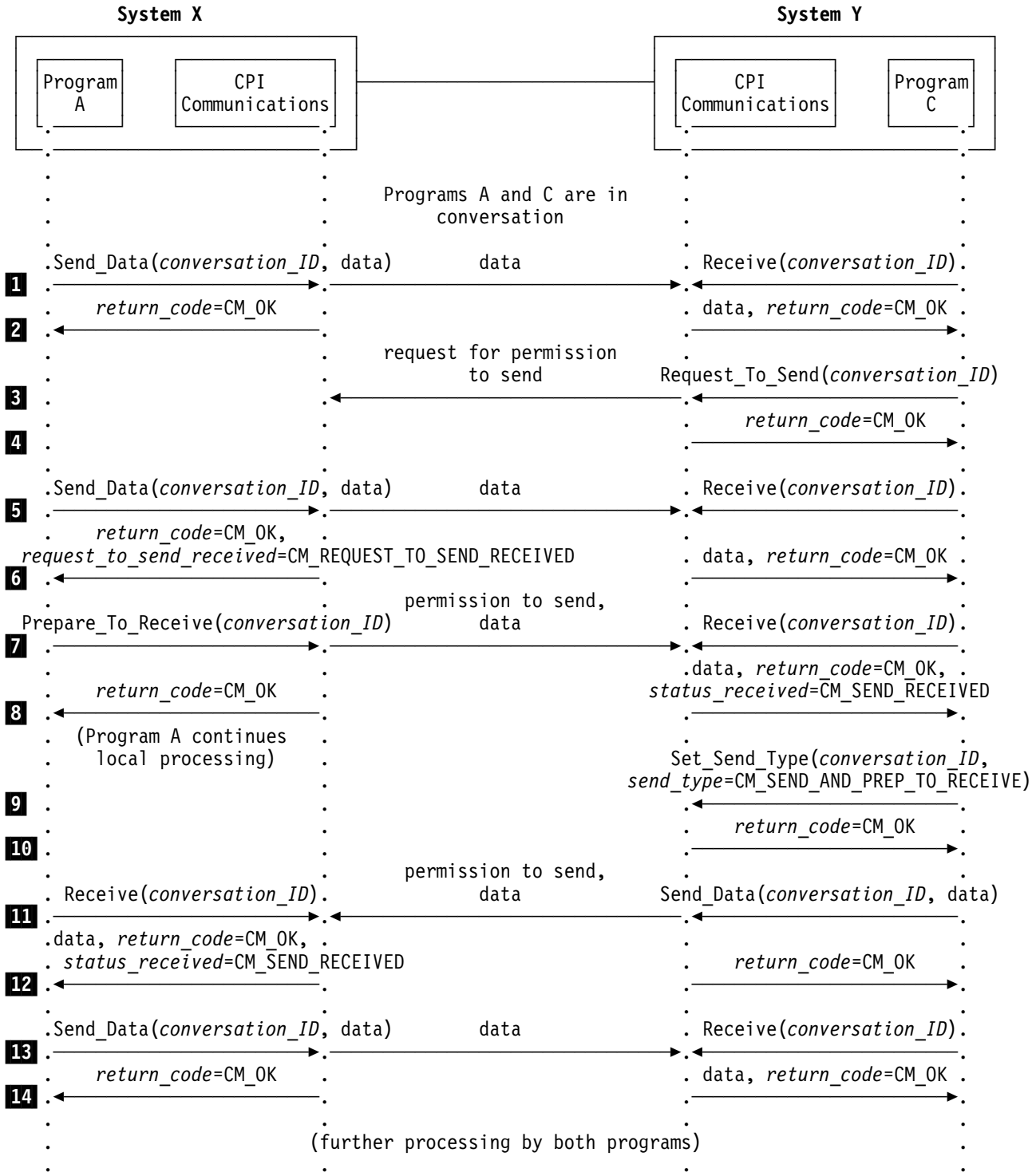


Figure 10. Confirmation of Data

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 11 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 52 describes a more complicated use of Send_Error.

The steps shown in Figure 11 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 system sends control information to 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 System X knows that a Send_Error has been issued at System Y (the control information sent in Step 3). After 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.

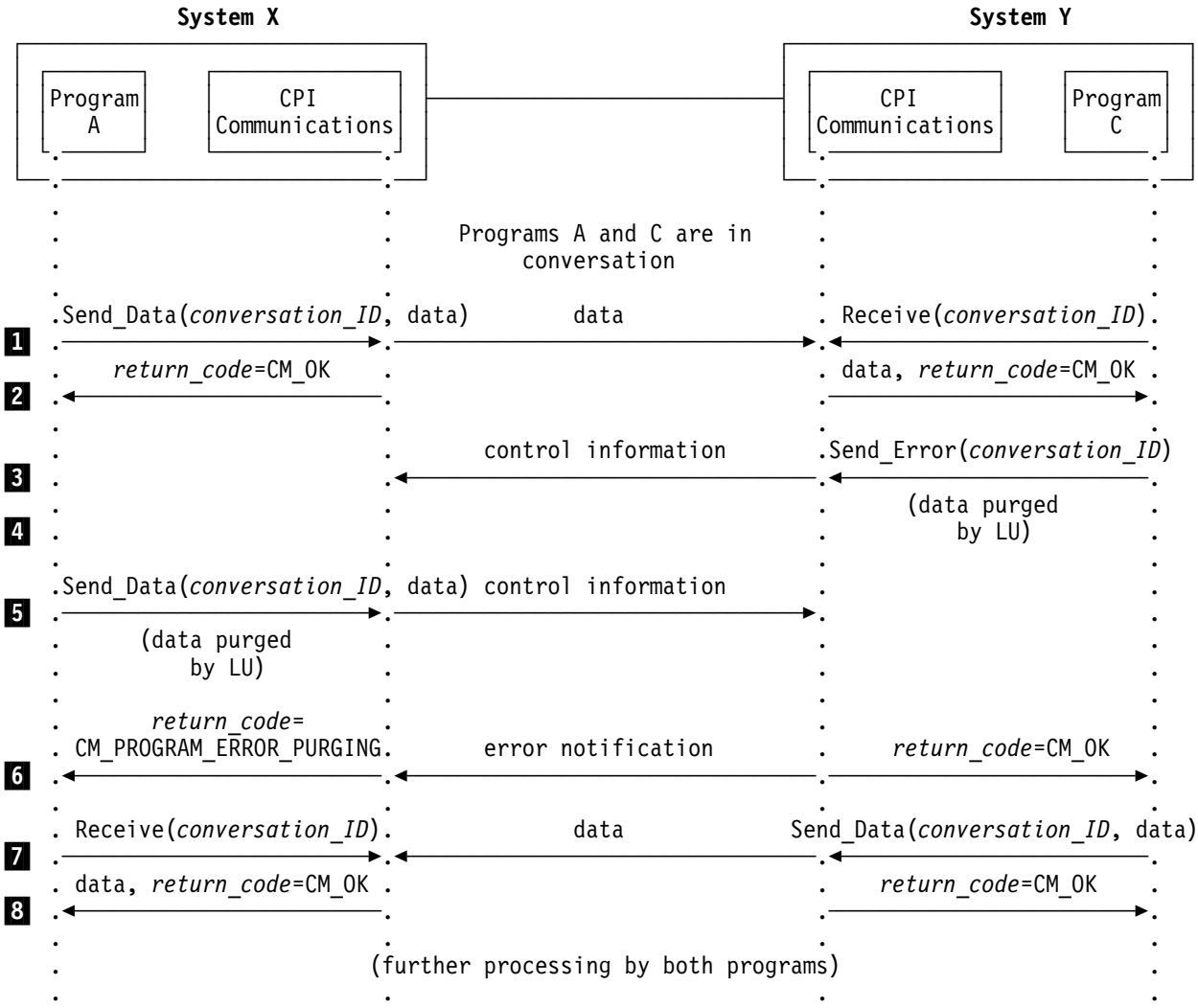


Figure 11. Reporting Errors

Example 7: Error Direction and Send-Pending State

Figure 12 on page 53 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 12 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 Program C did not receive send control with the data and, consequently, the conversation did not enter **Send-Pending** state. The *error_direction* characteristic is relevant only when the conversation is in **Send-Pending** state.
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.

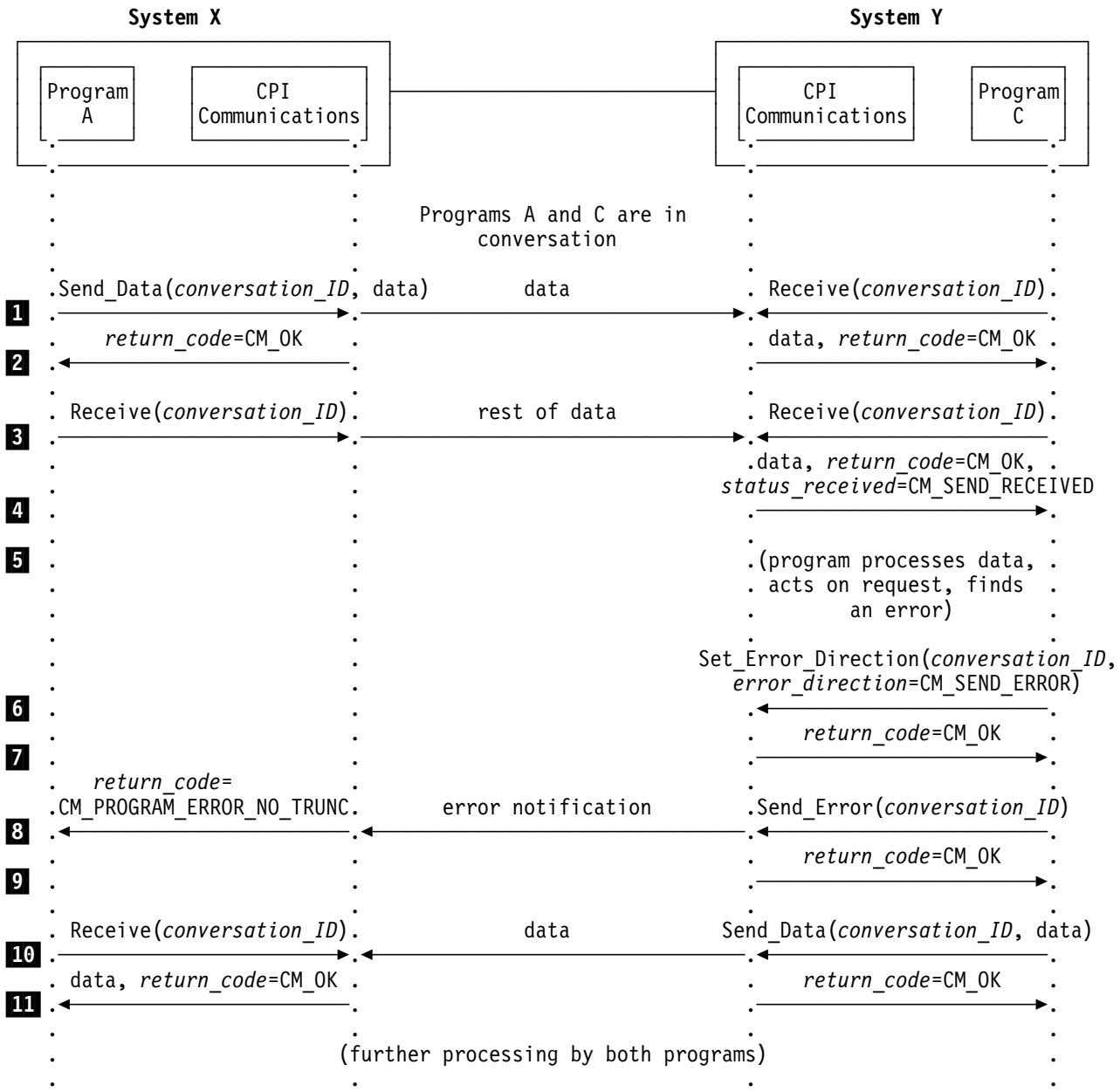


Figure 12. Error Direction and Send-Pending State

Examples 8–10 show the use of CPI Communications with a resource recovery interface. Because resource recovery interfaces are 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.

These examples use CPI Communications with the SAA resource recovery interface. The acronym **RR/SPM** is used in the following examples to represent the SAA resource recovery interface functioning in conjunction with a sync point manager. The resource recovery call names and return codes may be different for other resource recovery interfaces.

Example 8: Sending Program Issues a Commit

This example shows a program sending data on a protected conversation and issuing an SAA resource recovery 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 the calls of a resource recovery interface to manage the changes made to protected resources.

The steps shown in Figure 13 are:

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 an SAA resource recovery 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 an SAA resource recovery 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.

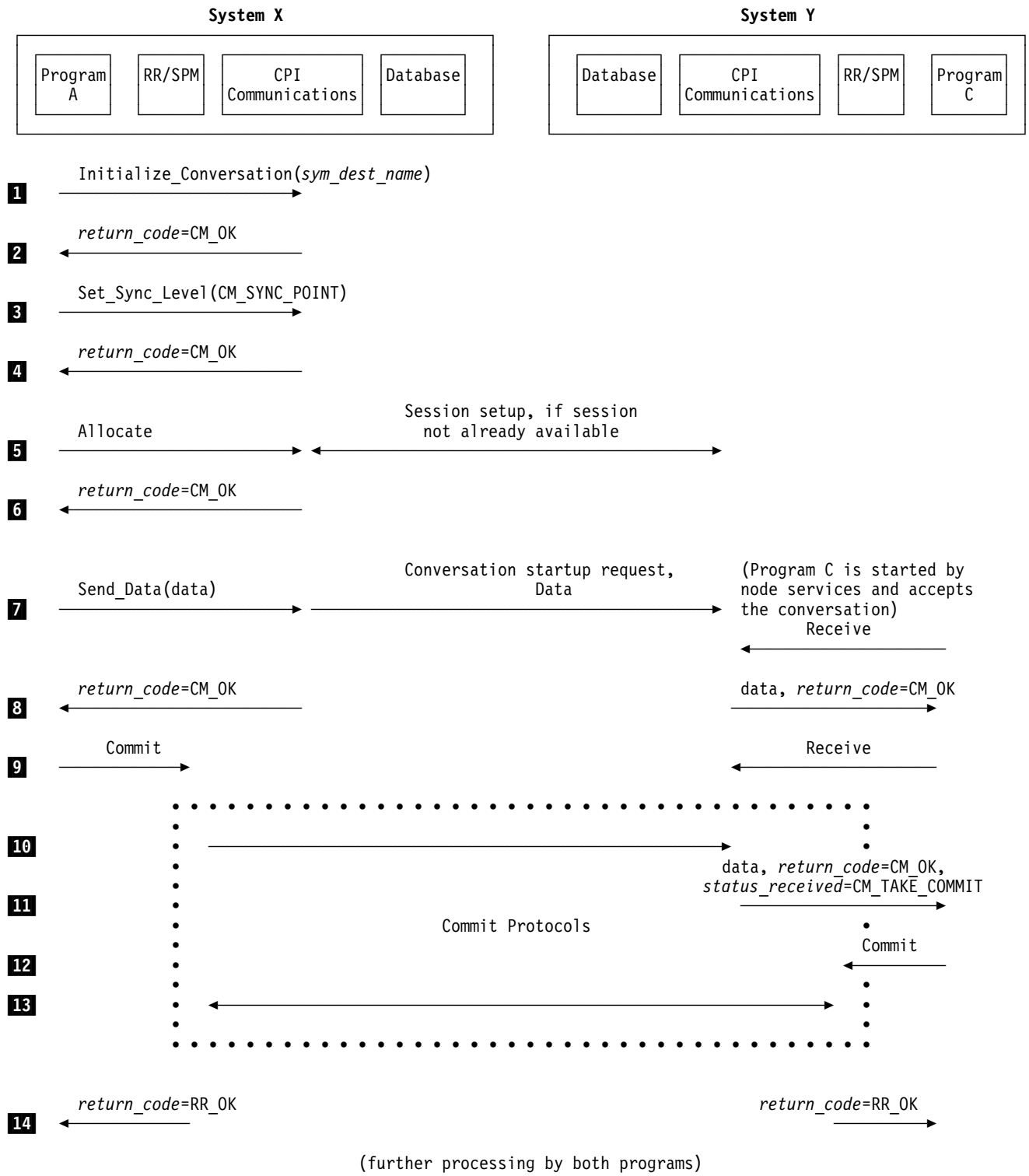


Figure 13. Establishing a Protected Conversation and Issuing a Successful Commit

Example 9: A Successful Commit with Conversation State Change

Figure 14 on page 57 shows a successful Commit with a conversation state change.

The steps shown in Figure 14 are:

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 an SAA resource recovery 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 an SAA resource recovery 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 the commit is unsuccessful and responses indicating backout are received on the commit calls by Programs A and C, the conversation states for Programs A and C are reset to their values at the time of the last sync point. The program can retrieve the current conversation state by using the CPI Communications Extract_Conversation_State call.

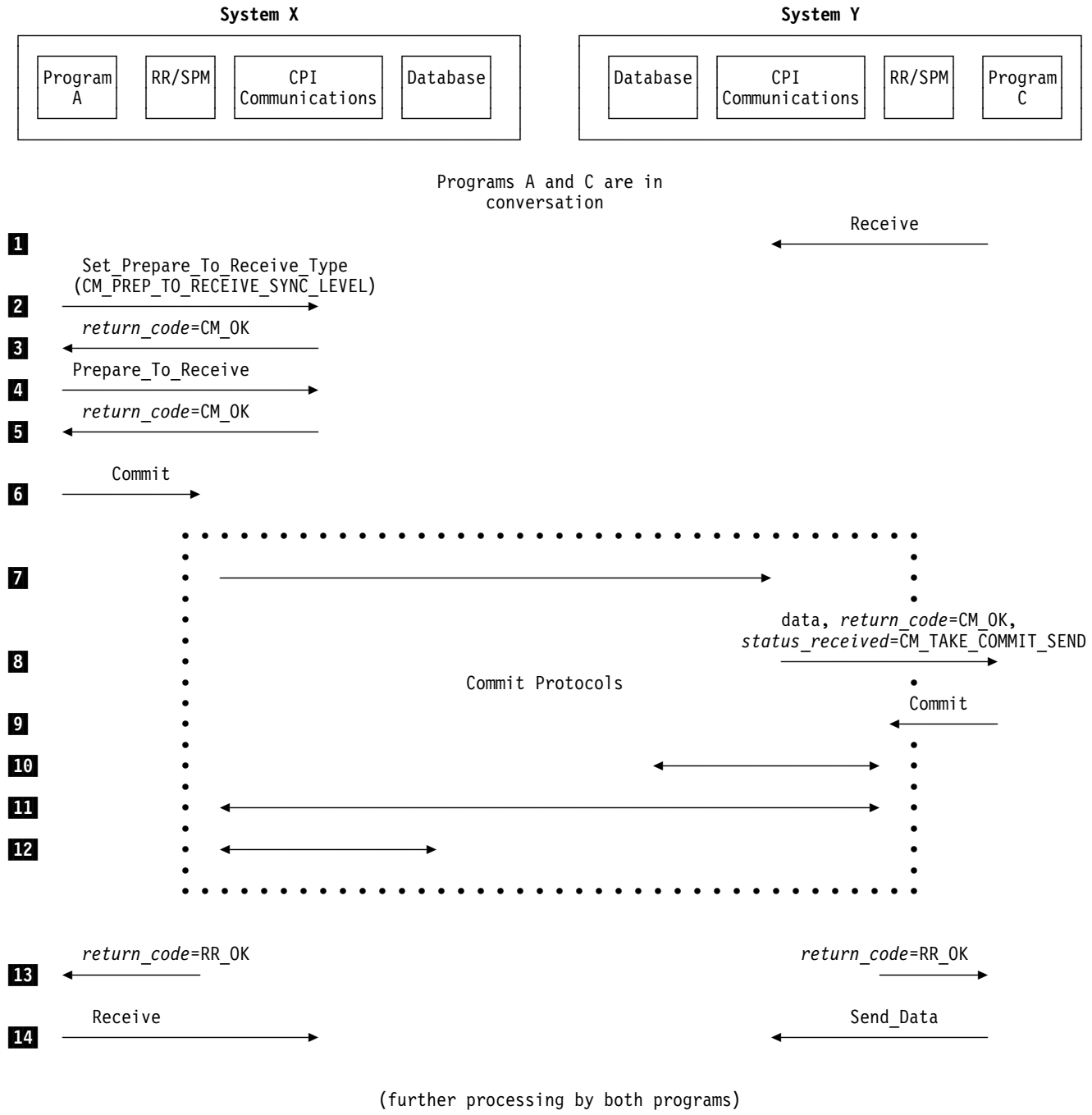


Figure 14. A Successful Commit with Conversation State Change

Example 10: Conversation Deallocation before the Commit Call

Figure 15 on page 59 shows a commit call issued after a CPI Communications Deallocate call.

The steps shown in Figure 15 are:

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 an SAA resource recovery 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 an SAA resource recovery 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 the commit is unsuccessful and responses indicating backout are received on the commit calls by Programs A and C, the conversation is not deallocated. The conversation states for Programs A and C are reset to their values at the time of the last sync point. The program can retrieve the current conversation state by using the CPI Communications Extract_Conversation_State call.

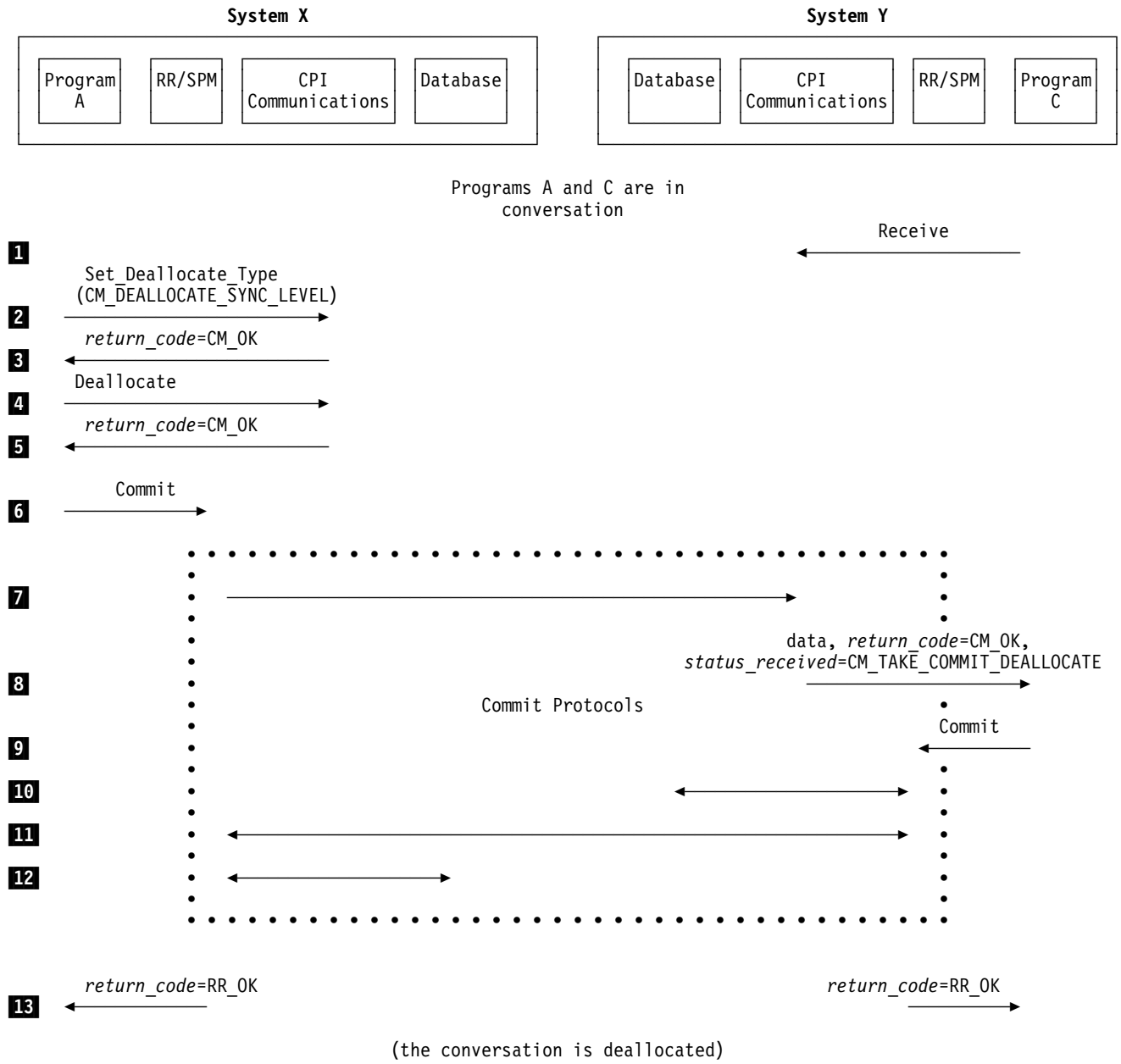


Figure 15. Conversation Deallocation Precedes the Commit Call

Example 11: Accepting Multiple Conversations Using Blocking Calls

Figure 16 on page 61 shows an example of a program that uses blocking calls to accept multiple incoming conversations.

The steps shown in Figure 16 are:

Step	Description
1	Program C is started as the result of a local operation and informs node services that it is ready to accept conversation startup requests for a program named "PAYROLL" by issuing the Specify_Local_TP_Name (CMSLTP) call.
2	Program C initializes the conversation on the accepting side by issuing the Initialize_For_Incoming call. Upon successful completion of this call, the conversation is in **Initialize-Incoming** state.
3	Program C accepts the incoming conversation with the Accept_Incoming call. The *conversation_ID* returned on the Initialize_For_Incoming call is supplied on the Accept_Incoming call. This call is blocked until the conversation startup request arrives. The *processing_mode* characteristic is initialized to the default value of CM_BLOCKING by the Initialize_For_Incoming call.
4	Program A uses the Initialize_Conversation call to initialize conversation characteristics for an outgoing conversation to Program C. In this example, the TP name characteristic is set to "PAYROLL".
5	Program A allocates the conversation, supplying the *conversation_ID* returned by the Initialize_Conversation call. In this example, the conversation startup request is sent as part of the Allocate processing.
	When System Y receives the conversation startup request, the Accept_Incoming call completes. A new context is created, and the conversation is assigned to that context. Node services sets Program C's current context to the new context.
6 and 7	Program C is ready to accept a second conversation, and it issues the Initialize_For_Incoming and Accept_Incoming calls. Again, the Accept_Incoming call is blocked until a conversation startup request arrives at System Y.
8 and 9	Program B on System Z initializes and allocates a conversation to "PAYROLL".
	When System Y receives the conversation startup request, the Accept_Incoming call completes with a return code of CM_OK. Another new context is created and the new conversation is assigned to that context. Node services sets Program C's current context to the new context.

Note: In steps 2, 3, 6, and 7, the program could use the Accept_Conversation call instead of issuing the Initialize_For_Incoming and Accept_Incoming calls.

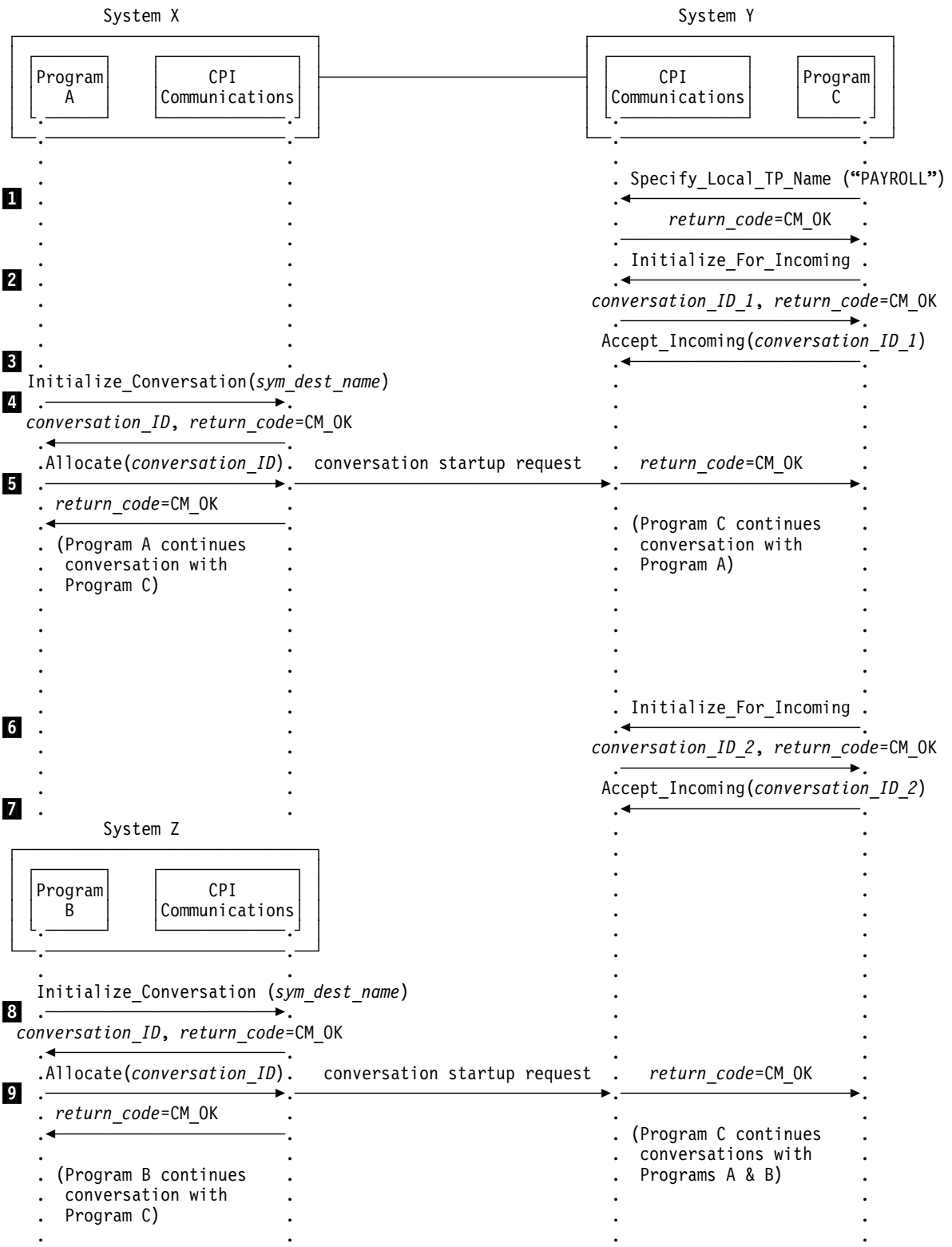


Figure 16. Accepting Multiple Conversations Using Blocking Calls

Example 12: Accepting Multiple Conversations Using Non-Blocking Calls

Figure 17 on page 63 shows an example of a program that uses non-blocking calls to accept multiple incoming conversations.

The steps shown in Figure 17 are:

Step	Description
1	Program C is started as the result of a local operation and informs node services that it is ready to accept conversation startup requests for a program named “PAYROLL” by issuing the Specify_Local_TP_Name call.
2	Program C prepares for an incoming conversation by issuing the Initialize_For_Incoming call. Upon successful completion of this call, the conversation is in **Initialize-Incoming** state.
	Note: The Initialize_For_Incoming call is required in this case since the conversation will be accepted in a non-blocking processing mode. The Accept_Conversation call cannot be used because it is a blocking call. A *conversation_ID* is needed to set the processing mode and none is available prior to issuing the Accept_Conversation call.
3 and 4	Program C sets the processing mode for the conversation to non-blocking and issues the Accept_Incoming call. Since no conversation is currently available, CM_OPERATION_INCOMPLETE is returned.
5	Program C waits for an incoming conversation with the Wait_For_Conversation call.
6	Program A prepares to allocate a conversation by issuing Initialize_Conversation to initialize the conversation characteristics. In this example, the TP name characteristic is set to “PAYROLL”.
7	Program A allocates a conversation. In this example, the conversation startup request is sent as part of the Allocate processing.
	When System Y receives the conversation startup request, the Wait_For_Conversation call completes, returning *conversation_ID_1*. A new context is created, and the conversation is assigned to that context. The program's current context is not changed.
8	Program C issues another Initialize_For_Incoming call to prepare to accept a second incoming conversation.
9 and 10	The Set_Processing_Mode call is used to set the processing mode for the conversation to non-blocking prior to issuing the Accept_Incoming call. Since there is no conversation startup request to receive, the call completes with CM_OPERATION_INCOMPLETE.
11	Program C again issues a Wait_For_Conversation call to wait for activity on either *conversation_ID_1* or *conversation_ID_2*.
12	Program B on System Z initializes conversation characteristics in preparation for allocating a conversation. In this example, the TP name characteristic is set to “PAYROLL”.
13	Program B allocates a conversation to Program C. In this example, the conversation startup request is sent as part of the Allocate processing.
	When System Y receives the conversation startup request, the outstanding Wait_For_Conversation call completes, returning *conversation_ID_2*. Another new context is created and the new conversation is assigned to that context. The program's current context is not changed.

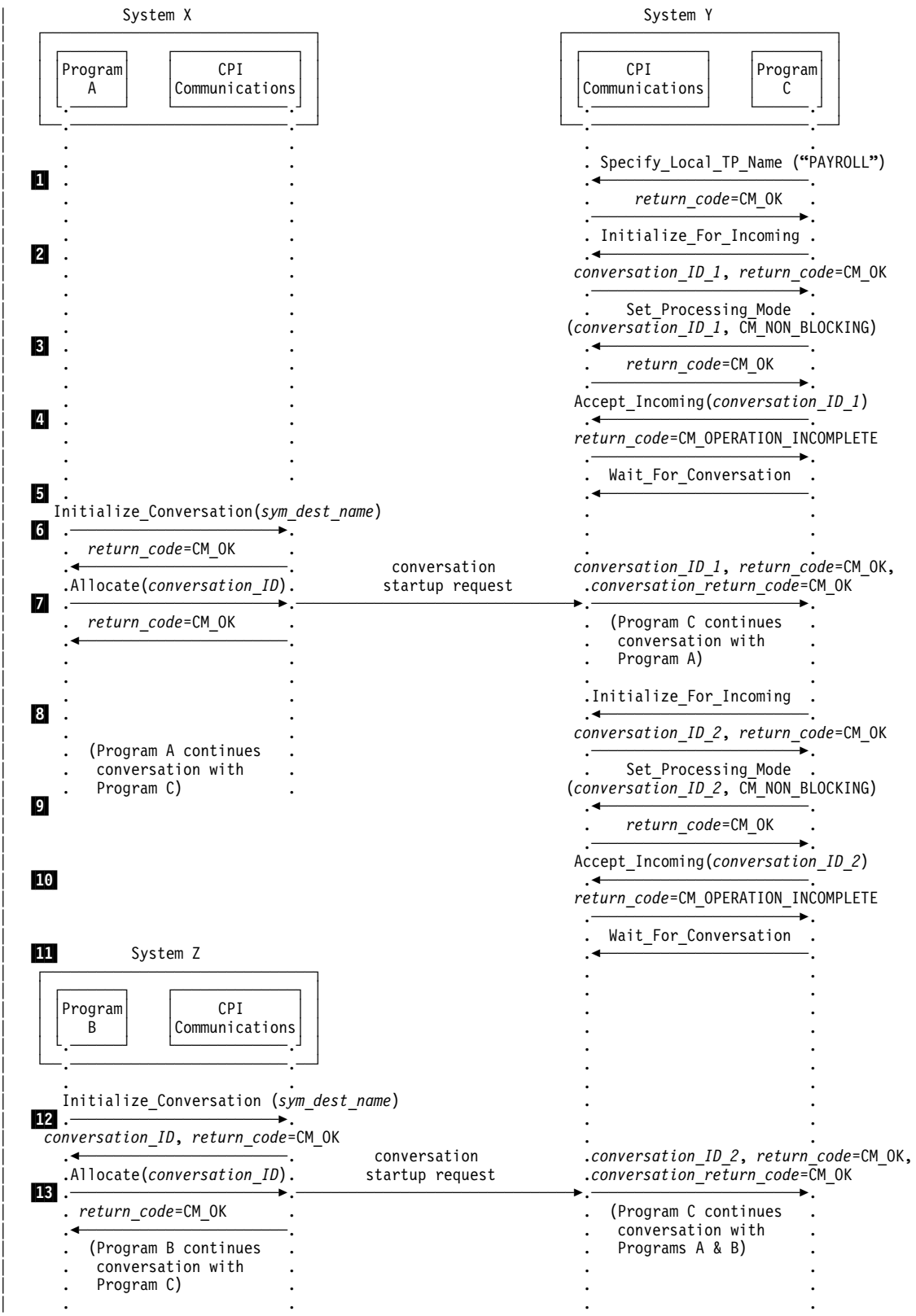


Figure 17. Accepting Multiple Conversations Using Non-Blocking Calls

Tutorial

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 66 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 25 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.

© Copyright IBM Corp. 1988, 1993

Call Syntax

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

- Application Generator (CSP)
- C
- COBOL
- FORTRAN
- PL/I
- REXX
- RPG.

In addition to the above programming languages, other languages may support CPI Communications calls in certain environments.

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 (CSP)

```
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)
```

REXX

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

RPG

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

Programming Language Considerations

This section describes some programming language considerations a programmer should keep in mind when writing and running a program that uses CPI Communications. Sample pseudonym files are provided in Appendix E, "Pseudonym Files" on page 231 for several programming languages. Customized pseudonym files or datasets for supported programming languages may be available on systems that implement CPI Communications.

Note: Some programming language processors (compilers and interpreters) may not support the asynchronous modification of a program's variables by another process. Use of non-blocking operations is not possible by programs using these language processors.

Application Generator

Cross System Product (CSP) is the implementing product for the SAA Application Generator common programming interface.

No special considerations apply to CPI Communications programs written in CSP.

C

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

- 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 calls:

- 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 calls:

- The EXTERNAL statement may be required for each CPI Communications call that is issued, 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 calls:

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

REXX

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

- REXX programs must use the ADDRESS CPICOMM statement to access CPI Communications calls. These calls are not accessible through the REXX CALL statement interface.
- Character strings returned by CPI Communications calls are stored in variables with the maximum allowable length. (Maximum lengths are shown in Appendix A, "Variables and Characteristics" on page 191.) 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 calls:

- 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 176 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 66 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 198.

- Table 5 on page 192 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 70. Key-topic discussions and where they occur are:

- “Naming Conventions—Calls and Characteristics, Variables and Values” on page 26 describes the naming conventions used throughout the book.
- “Data Buffering and Transmission” on page 43 discusses program control over data transmission.
- “Usage Notes” of “Request_To_Send (CMRTS)” on page 135 discusses how a conversation enters **Receive** state.
- “Usage Notes” of “Send_Data (CMSEND)” on page 140 describes the use of logical records and LL fields on basic conversations.

Reference Section

Pseudonym	Call	Description	Page
Accept_Conversation	CMACCP	Used by a program to accept an incoming conversation.	73
Accept_Incoming	CMACCI	Used by a program to accept an incoming conversation previously initialized with the Initialize_For_Incoming call.	75
Allocate	CMALLC	Used by a program to establish a conversation.	77
Cancel_Conversation	CMCANC	Used by a program to end a conversation immediately.	80
Confirm	CMCFM	Used by a program to send a confirmation request to its partner.	82
Confirmed	CMCFMD	Used by a program to send a confirmation reply to its partner.	85
Convert_Incoming	CMCNVI	Used by a program to change the encoding of a character string from EBCDIC to the local encoding used by the program.	87
Convert_Outgoing	CMCNVO	Used by a program to change the encoding of a character string from the local encoding used by the program to EBCDIC.	89
Deallocate	CMDEAL	Used by a program to end a conversation.	91
Extract_Conversation_Context	CMECTX	Used by a program to extract the *context_ID* for a conversation.	96
Extract_Conversation_State	CMECS	Used by a program to view the current state of a conversation.	98
Extract_Conversation_Type	CMECT	Used by a program to view the current *conversation_type* conversation characteristic.	100
Extract_Maximum_Buffer_Size	CMEMBS	Used by a program to extract the maximum buffer size supported by the system.	102
Extract_Mode_Name	CMEMN	Used by a program to view the current *mode_name* conversation characteristic.	103
Extract_Partner_LU_Name	CMEPLN	Used by a program to view the current *partner_LU_name* conversation characteristic.	105
Extract_Security_User_ID	CMESUI	Used by a program to view the current *security_user_ID* conversation characteristic.	107
Extract_Sync_Level	CMESL	Used by a program to view the current *sync_level* conversation characteristic.	109
Extract_TP_Name	CMETPN	Used by a program to determine the *TP_name* characteristic's value for a given conversation.	111
Flush	CMFLUS	Used by a program to flush the LU's send buffer.	113
Initialize_Conversation	CMINIT	Used by a program to initialize the conversation characteristics for an outgoing conversation.	115

Pseudonym	Call	Description	Page
Initialize_For_Incoming	CMINIC	Used by a program to initialize the conversation characteristics for an incoming conversation.	117
Prepare_To_Receive	CMPTR	Used by a program to change a conversation from **Send** to **Receive** state in preparation to receive data.	119
Receive	CMRCV	Used by a program to receive data.	123
Release_Local_TP_Name	CMRLTP	Used by a program to release a name.	133
Request_To_Send	CMRTS	Used by a program to notify its partner that it would like to send data.	135
Send_Data	CMSEND	Used by a program to send data.	137
Send_Error	CMSERR	Used by a program to notify its partner of an error that occurred during the conversation.	143
Set_Conversation_Security_Password	CMSCSP	Used by a program to set the *security_password* conversation characteristic.	149
Set_Conversation_Security_Type	CMSCST	Used by a program to set the *conversation_security_type* conversation characteristic.	151
Set_Conversation_Security_User_ID	CMSCSU	Used by a program to set the *security_user_ID* conversation characteristic.	153
Set _Conversation_Type	CMSCT	Used by a program to set the *conversation_type* conversation characteristic.	155
Set_Deallocate_Type	CMSDT	Used by a program to set the *deallocate_type* conversation characteristic.	157
Set_Error_Direction	CMSED	Used by a program to set the *error_direction* conversation characteristic.	160
Set_Fill	CMSF	Used by a program to set the *fill* conversation characteristic.	162
Set_Log_Data	CMSLD	Used by a program to set the *log_data* conversation characteristic.	164
Set_Mode_Name	CMSMN	Used by a program to set the *mode_name* conversation characteristic.	166
Set_Partner_LU_Name	CMSPLN	Used by a program to set the *partner_LU_name* conversation characteristic.	168
Set_Prepare_To_Receive_Type	CMSPTR	Used by a program to set the *prepare_to_receive_type* conversation characteristic.	170
Set_Processing_Mode	CMSPM	Used by a program to set the *processing_mode* conversation characteristic.	172
Set_Receive_Type	CMSRT	Used by a program to set the *receive_type* conversation characteristic.	174
Set_Return_Control	CMSRC	Used by a program to set the *return_control* conversation characteristic.	176

Reference Section

Pseudonym	Call	Description	Page
Set_Send_Type	CMSST	Used by a program to set the *send_type* conversation characteristic.	178
Set_Sync_Level	CMSSL	Used by a program to set the *sync_level* conversation characteristic.	180
Set_TP_Name	CMSTPN	Used by a program to set the *TP_Name* conversation characteristic.	182
Specify_Local_TP_Name	CMSLTP	Used by a program to associate a name with itself.	184
Test_Request_To_Send_Received	CMTRTS	Used by a program to determine whether or not the remote program is requesting to send data.	185
Wait_For_Conversation	CMWAIT	Used by a program to wait for the completion of any outstanding operation.	187

Accept_Conversation (CMACCP)

The Accept_Conversation (CMACCP) call accepts an incoming conversation. Like Initialize_Conversation, 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 one of the following:
  - No incoming conversation exists.
  - No name is associated with the program. A program associates a name with itself by issuing the Specify_Local_TP_Name call.
- 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. There may be a system-defined limit on the number of conversations that a program can accept or allocate, but CPI Communications imposes no limit.
3. For a list of the conversation characteristics that are initialized when the Accept_Conversation call completes successfully, see Table 3 on page 20.
4. CPI Communications makes incoming conversations available to programs based upon names that are associated with the program. Specifically, those names associated with the program at the time the Accept_Conversation call is issued are used to satisfy that Accept_Conversation call. These names come either from locally defined information or from execution of the

Specify_Local_TP_Name call. An implementation may place restrictions on the actions that a program may take before issuing Accept_Conversation in order to properly identify programs with associated names.

5. Accept_Conversation always functions as if the *processing_mode* were set to CM_BLOCKING. A program that must be able to accept incoming conversations in a non-blocking mode should use the Initialize_For_Incoming and Accept_Incoming calls. The processing mode for the conversation can be set to CM_NON_BLOCKING prior to issuing Accept_Incoming.
6. A new context is created as a result of the successful completion of the Accept_Conversation call, and the conversation is assigned to the new context. The program's current context is set to the new context by node services.

Related Information

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

"Example 1: Data Flow in One Direction" on page 37 shows an example program flow using the Accept_Conversation call.

"Initialize_Conversation (CMINIT)" on page 115 describes how the conversation characteristics are initialized for the program that allocates the conversation.

Accept_Incoming (CMACCI)

A program uses the Accept_Incoming (CMACCI) call to accept an incoming conversation that has previously been initialized with the Initialize_For_Incoming call and to complete the initialization of the conversation characteristics.

Before issuing the Accept_Incoming call, a program has the option of issuing the following call to specify whether the Accept_Incoming call, and all subsequent calls, are to be processed in a blocking or non-blocking mode:

CALL CMSPM – Set_Processing_Mode

Format

```
CALL CMACCI(conversation_ID,
            return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier of a conversation that has been initialized for an incoming 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_OPERATION_INCOMPLETE
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  – The conversation is not in **Initialize-Incoming** state.
  – The *processing_mode* is set to CM_BLOCKING and no incoming conversation exists.
  – No name is associated with the program. A program associates a name with itself by issuing the Specify_Local_TP_Name call.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

When *return_code* is set to CM_OK, the conversation enters **Receive** state.

Usage Notes

1. The Accept_Incoming call can be used only when an Initialize_For_Incoming call has already completed.
2. When Accept_Incoming successfully completes, CPI Communications initializes those conversation characteristics that use values from the conversation startup request. See Table 3 on page 20 for a list of the conversation characteristics and how they are set by Initialize_For_Incoming and Accept_Incoming.

3. The program can ensure that it is not placed in a wait state by setting the *processing_mode* to CM_NON_BLOCKING prior to issuing the Accept_Incoming call.
4. There may be a system-defined limit on the number of conversations that a program can accept or allocate, but CPI Communications imposes no limit.
5. CPI Communications makes incoming conversations available to programs based upon names that are associated with the program. Specifically, those names associated with the program at the time the Accept_Incoming call is issued are used to satisfy that Accept_Incoming call. These names come either from locally defined information or from execution of the Specify_Local_TP_Name call. An implementation may place restrictions on the actions that a program may take before issuing Accept_Incoming in order to properly identify programs with associated names.
6. A new context is created as a result of the successful completion of the Accept_Incoming call, and the conversation is assigned to the new context. If Accept_Incoming completes successfully with *return_code* set to CM_OK, the program's current context is set to the new context by node services. If Accept_Incoming is issued with *processing_mode* set to CM_NON_BLOCKING and gets the CM_OPERATION_INCOMPLETE return code, the program's current context is not changed if the Accept_Incoming call operation subsequently completes successfully as a result of the Wait_For_Conversation call.

Related Information

"Initialize_For_Incoming (CMINIC)" on page 117 describes how the *conversation_ID* supplied on Accept_Incoming is assigned.

"Set_Processing_Mode (CMSPM)" on page 172 describes the processing mode conversation characteristic.

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

"Example 11: Accepting Multiple Conversations Using Blocking Calls" on page 60 and "Example 12: Accepting Multiple Conversations Using Non-Blocking Calls" on page 62 show example program flows using the Initialize_For_Incoming and Accept_Incoming calls.

The calls beginning with "Extract" in this chapter are used to examine conversation characteristics established by the Accept_Incoming call.

Allocate (CMALLC)

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:

CALL CMSCSP – Set_Conversation_Security_Password
CALL CMSCST – Set_Conversation_Security_Type
CALL CMSCSU – Set_Conversation_Security_User_ID
CALL CMSCT – Set_Conversation_Type
CALL CMSMN – Set_Mode_Name
CALL CMSPLN – Set_Partner_LU_Name
CALL CMSPM – Set_Processing_Mode
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_OPERATION_INCOMPLETE
- 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 the local program does not have the authority to specify. For example, SNASVCMG requires special authority with LU 6.2.
  - The *TP_name* characteristic (set from side information or by Set_TP_Name) specifies a transaction program name that the local program does not have the appropriate authority to allocate a conversation to. For example, SNA service programs require special authority with LU 6.2. (For more information, see “SNA Service Transaction Programs” on page 227.)
  - 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 as being valid.
  - The *conversation_security_type* characteristic is CM_SECURITY_PROGRAM, and the *security_password* characteristic or the *security_user_ID* characteristic (set from side information or by SET calls), or both, are null.
- 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 conversation's context is in the **Backout-Required** condition. New protected conversations cannot be allocated for a context when it is in this condition.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_OPERATION_NOT_ACCEPTED
- 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 system's failure to obtain a session for the conversation is reported on the Allocate call. An allocation error resulting from the remote system'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 system and the remote system, if such a session does not already exist.
3. Depending on the circumstances, the local system can send the conversation allocation request to the remote system as soon as it allocates a session for the conversation. The local system 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 system 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, based on the values of the security characteristics. See "Conversation Security" on page 24 for more information on conversation security.
6. When *processing_mode* is set to CM_NON_BLOCKING and *return_control* is set to CM_IMMEDIATE, the call completes immediately. If a session is not available, *return_code* is set to CM_UNSUCCESSFUL.
7. When a program allocates a conversation, the program's current context at the time the Allocate call is issued becomes the new conversation's context. If the program has multiple contexts (for example, as a result of accepting multiple conversations), it must ensure that the current context is set to the appropriate context before allocating the new conversation.

Related Information

"Example 1: Data Flow in One Direction" on page 37 shows an example program flow using the Allocate call.

"SNA Service Transaction Programs" on page 227 discusses SNA service transaction programs.

"Data Buffering and Transmission" on page 43 discusses control methods for data transmission.

"Set_Return_Control (CMSRC)" on page 176 discusses the *return_control* characteristic.

"Set_Conversation_Type (CMSCT)" on page 155 discusses the *conversation_type* characteristic.

"Extract_Conversation_Context (CMECTX)" on page 96 discusses the *context_ID* characteristic.

"Contexts and Context Management" on page 17 defines contexts and discusses how application programs can manage multiple contexts.

Cancel_Conversation (CMCANC)

A program uses Cancel_Conversation (CMCANC) to end a conversation immediately. Cancel_Conversation can be issued in blocking or non-blocking mode, and regardless of whether there is an outstanding operation on the conversation.

Cancel_Conversation results in the immediate termination of any outstanding operation on the specified conversation. No guarantees are given on the results of the outstanding operation. For example, when a Cancel_Conversation call has been issued while a non-blocking Send_Data call is outstanding, no statement may be made about how much data was actually moved from the application buffer or about the contents of the *request_to_send* parameter.

Format

```
CALL CMCANC(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_PARAMETER_CHECK
  This value 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 **Reset** state.

Usage Notes

1. From the perspective of the local program, the conversation is terminated immediately. However, CPI Communications may not be able to notify the remote program until a later time.
2. The remote program will be notified of the termination of the conversation with the CM_DEALLOCATED_ABEND return code or, if the conversation has *sync_level* set to CM_SYNC_POINT and backout is required, with the CM_DEALLOCATED_ABEND_BO return code.

   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 or CM_DEALLOCATED_NORMAL_BO return code is reported instead of CM_DEALLOCATED_ABEND or CM_DEALLOCATED_ABEND_BO, respectively. See “Send_Error (CMSERR)” on page 143 for a complete discussion.

3. Program-supplied log data is not sent to the remote system as a result of a Cancel_Conversation call.
4. When Cancel_Conversation is issued for a protected conversation, the conversation's context may be placed in the **Backout-Required** condition.

Related Information

"Wait_For_Conversation (CMWAIT)" on page 187 describes the normal completion of non-blocking operations.

"Non-Blocking Operations" on page 22 discusses the use of non-blocking operations.

Confirm (CMCFM)

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_OPERATION_INCOMPLETE
- 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 (basic conversations only)
- CM_DEALLOCATED_ABEND_TIMER (basic conversations only)
- CM_SVC_ERROR_PURGING (basic conversations only)
- 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 conversation's context is in the **Backout-Required** condition. The Confirm call is not allowed for this conversation while its context 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_OPERATION_NOT_ACCEPTED
- 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 (basic conversations only)
  - CM_DEALLOCATED_ABEND_TIMER_BO (basic conversations only)
  - 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 system is flushed as a result of this call.

Related Information

"Confirmed (CMCFMD)" on page 85 provides information on the remote program's reply to the Confirm call.

"Request_To_Send (CMRTS)" on page 135 provides a complete discussion of the *request_to_send_received* parameter.

"Set_Sync_Level (CMSSL)" on page 180 explains how programs specify the level of synchronization processing.

"Example 4: Validation and Confirmation of Data Reception" on page 46 shows an example program using the Confirm call.

Confirmed (CMCFMD)

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_OPERATION_INCOMPLETE
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The conversation is not in **Confirm**, **Confirm-Send**, or **Confirm-Deallocate** state.
  - For a conversation with *sync_level* set to CM_SYNC_POINT, the conversation's context is in the **Backout-Required** condition. The Confirmed call is not allowed for this conversation while its context is in this condition.
- CM_PROGRAM_PARAMETER_CHECK
  This return code indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_OPERATION_NOT_ACCEPTED
- 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 system) 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 82 provides more information on the Confirm call.

"Receive (CMRCV)" on page 123 provides more information on the *status_received* parameter.

"Set_Sync_Level (CMSSL)" on page 180 explains how programs specify the level of synchronization processing.

"Example 4: Validation and Confirmation of Data Reception" on page 46 shows an example program using the Confirmed call.

Convert_Incoming (CMCNVI)

The Convert_Incoming (CMCNVI) call is used to change the encoding of a character string from EBCDIC to the local encoding used by the program.

Format

```
CALL CMCNVI(buffer,
            buffer_length,
            return_code)
```

Parameters

buffer *(input) (output)*
Specifies the buffer containing the string to be converted. The contents of the string will be replaced by the results of the conversion.

buffer_length *(input)*
Specifies the number of characters in the string to be converted.

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 *buffer_length* is invalid for the range permitted by the implementation.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call causes no state changes.

Usage Notes

1. The EBCDIC hexadecimal codes, specified in Table 6 on page 195, represent the encoding for the data transmitted across the network. The Convert_Incoming call converts the EBCDIC hexadecimal codes to the corresponding local representation of the data.
2. Convert_Incoming converts data on a character-by-character basis. Since the program may use character values beyond those defined in Table 6 on page 195, care must be taken in the use of Convert_Incoming in that it may generate implementation-dependent results if applied to a string which contains such values.
3. A program may be written to be independent of the encoding (such as ASCII or EBCDIC) of the partner program by sending and receiving EBCDIC data records with the help of the Convert_Outgoing and Convert_Incoming calls. The sending program calls Convert_Outgoing to convert the data record to EBCDIC before sending it. The receiving program calls Convert_Incoming to convert the EBCDIC data record to the appropriate encoding for its environment.
4. The Convert_Incoming call is a null operation if the receiving program uses EBCDIC encoding.

Related Information

"Data Conversion" on page 24 provides information about data conversion and the Convert_Incoming call.

"Convert_Outgoing (CMCNVO)" on page 89 provides information about the Convert_Outgoing call.

Convert_Outgoing (CMCNVO)

The Convert_Outgoing (CMCNVO) call is used to change the encoding of a character string to EBCDIC from the local encoding used by the program.

Format

```
CALL CMCNVO(buffer,
            buffer_length,
            return_code)
```

Parameters

buffer *(input) (output)*
Specifies the buffer containing the string to be converted. The contents of the string will be replaced by the results of the conversion.

buffer_length *(input)*
Specifies the number of characters in the string to be converted.

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 *buffer_length* is invalid for the range permitted by the implementation.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call causes no state changes.

Usage Notes

1. The EBCDIC hexadecimal codes, specified in Table 6 on page 195, represent the encoding for the data transmitted across the network. The Convert_Outgoing call converts the data supplied by the program from the local encoding to the corresponding EBCDIC hexadecimal codes.
2. Convert_Outgoing converts data on a character-by-character basis. Since the program may use character values beyond those defined in Table 6 on page 195, care must be taken in the use of Convert_Outgoing in that it may generate implementation-dependent results if applied to a string which contains such values.
3. A program may be written to be independent of the encoding (such as ASCII or EBCDIC) of the partner program by sending and receiving EBCDIC data records with the help of the Convert_Outgoing and Convert_Incoming calls. The sending program calls Convert_Outgoing to convert the data record to EBCDIC before sending it. The receiving program calls Convert_Incoming to convert the EBCDIC data record to the appropriate encoding for its environment.
4. The Convert_Outgoing call is a null operation if the sending program uses EBCDIC encoding.

Related Information

"Data Conversion" on page 24 provides information about data conversion and the Convert_Outgoing call.

"Convert_Incoming (CMCNVI)" on page 87 provides information about the Convert_Incoming call.

Deallocate (CMDEAL)

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 a resource recovery 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_OPERATION_INCOMPLETE
- 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_OPERATION_NOT_ACCEPTED
- 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_OPERATION_INCOMPLETE
- 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 (basic conversations only)
- CM_DEALLOCATED_ABEND_TIMER (basic conversations only)
- 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_OPERATION_NOT_ACCEPTED
- 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 a resource recovery 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 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 223 shows how SAA resource recovery calls affect CPI Communications conversation states.
- CM_OPERATION_INCOMPLETE
- 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 conversation's context is in the **Backout-Required** condition.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_OPERATION_NOT_ACCEPTED
- 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 system's send buffer. If *deallocate_type* is CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is CM_SYNC_POINT, the local system's send buffer will not be flushed until a resource recovery 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 system 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 143 for a complete discussion.

- CM_DEALLOCATED_ABEND_BO *return_code*
  This indicates that the partner program issued Deallocate with the *deallocate_type* set to CM_DEALLOCATE_ABEND and the conversation is protected.

  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_BO is purged and a CM_DEALLOCATED_NORMAL_BO *return_code* is reported instead of CM_DEALLOCATED_ABEND_BO. See "Send_Error (CMSERR)" on page 143 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 a resource recovery 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. The way abnormal deallocation is accomplished is implementation-specific.
6. When a Deallocate call is issued with *deallocate_type* set to CM_DEALLOCATE_ABEND and *sync_level* set to CM_SYNC_POINT, the conversation's context may be placed in the **Backout-Required** condition.
7. Note to Implementers: The following deallocation-related CM_PROGRAM_PARAMETER_CHECK error can occur, but is detected at Set_Deallocate_Type time:
   - *deallocate_type*(CM_DEALLOCATE_CONFIRM) is specified and the conversation is allocated with *sync_level*(CM_NONE).
   - *deallocate_type*(CM_DEALLOCATE_CONFIRM or CM_DEALLOCATE_FLUSH) is specified and the conversation is allocated with *sync_level*(CM_SYNC_POINT).
8. Note to Implementers: *deallocate_type*(CM_DEALLOCATE_ABEND) issued against a basic conversation implies the LU 6.2 protocol boundary return code of DEALLOCATE_ABEND_PROG. No separate parameter value is supported since a separate value would not provide any additional function. In LU 6.2, the sense data 08640000 maps to a return code of DEALLOCATE_ABEND for mapped conversations and to DEALLOCATE_ABEND_PROG for basic conversations. It does not appear useful to require the application to check for different return codes that have the same meaning depending on the conversation type.

Related Information

"Example 1: Data Flow in One Direction" on page 37 shows an example program flow using the Deallocate call.

"Set_Log_Data (CMSLD)" on page 164 discusses the *log_data* characteristic.

“Set_Deallocate_Type (CMSDT)” on page 157 discusses the *deallocate_type* characteristic and its possible values.

Extract_Conversation_Context (CMECTX)

Extract_Conversation_Context (CMECTX) is used by a program to extract the context identifier for a conversation.

Format

```
CALL CMECTX(conversation_ID,
            context_ID,
            context_ID_length,
            return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

context_ID *(output)*
Specifies the variable containing the context identifier for this conversation.

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

context_ID_length *(output)*
Specifies the variable containing the length of the returned *context_ID* parameter.

Note: Unless *return_code* is set to CM_OK, the value of *context_ID_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_PROGRAM_STATE_CHECK
  This value indicates that the conversation is in **Initialize** or **Initialize-Incoming** state.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. This call does not change the *context_ID* for the specified conversation.
2. A program may want to determine the context for a conversation prior to allocating a new conversation. The program would issue Extract_Conversation_Context to get the *context_ID* and then would use node services to set that context as the current context prior to issuing the Allocate call for the new conversation.

Related Information

"Contexts and Context Management" on page 17 defines contexts and discusses how application programs can manage multiple contexts.

Extract_Conversation_State (CMECS)

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
- CM_INITIALIZE_INCOMING_STATE

Note: Unless *return_code* is set to CM_OK, the value of *conversation_state* 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_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** or **Initialize-Incoming** state.
  - The conversation's context is in the **Backout-Required** condition.
  - The program is using protected resources that must be backed out.
- CM_OPERATION_NOT_ACCEPTED
- 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 a resource recovery backout operation.

Related Information

"Support for Resource Recovery Interfaces" on page 28 provides more information on using resource recovery interfaces.

Extract_Conversation_Type (CMECT)

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_PROGRAM_STATE_CHECK
  This value indicates that the *conversation_ID* specifies a conversation in **Initialize-Incoming** state.
- CM_OPERATION_NOT_ACCEPTED
- 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 155 provides more information on the *conversation_type* characteristic.

Extract_Maximum_Buffer_Size (CMEMBS)

A program uses the Extract_Maximum_Buffer_Size (CMEMBS) call to extract the maximum buffer size supported by the system.

Format

```
CALL CMEMBS(maximum_buffer_size,
            return_code)
```

Parameters

maximum_buffer_size *(output)*
Specifies the variable containing the maximum buffer size supported by the system.

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

State Changes

This call causes no state changes.

Usage Notes

This call can be used to find out the maximum buffer size supported by the system, when the maximum value is not known during program development. The value in *maximum_buffer_size* determines the largest amount of data the program can send in a Send_Data call. The largest amount of data that can be received by the program in a Receive call is:

- For a basic conversation
  - 32767, if the *fill* characteristic is set to CM_FILL_LL
  - The value of *maximum_buffer_size*, if the *fill* characteristic is set to CM_FILL_BUFFER
- For a mapped conversation
  - The value of *maximum_buffer_size*. The program should be aware that the CM_INCOMPLETE_DATA_RECEIVED value of the *data_received* parameter on the Receive call may be returned when the local maximum buffer size is less than the program partner's maximum buffer size.

Extract_Mode_Name (CMEMN)

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_PROGRAM_STATE_CHECK
  This value indicates that the *conversation_ID* specifies a conversation in **Initialize-Incoming** state.
- CM_OPERATION_NOT_ACCEPTED
- 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 166 and "Side Information" on page 11 provide further information on the *mode_name* characteristic.

"Automatic Conversion of Characteristics" on page 22 provides further information on the automatic conversion of the *mode_name* parameter.

Extract_Partner_LU_Name (CMEPLN)

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_PROGRAM_STATE_CHECK
  This value indicates that the *conversation_ID* specifies a conversation in **Initialize-Incoming** state.
- CM_OPERATION_NOT_ACCEPTED
- 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 168 and "Side Information" on page 11 provide more information on the *partner_LU_name* characteristic.

"Automatic Conversion of Characteristics" on page 22 provides further information on the automatic conversion of the *partner_LU_name* parameter.

Extract_Security_User_ID (CMESUI)

A program uses Extract_Security_User_ID (CMESUI) to extract the value of the *security_user_ID* characteristic for a given conversation.

Format

```
CALL CMESUI(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 containing the user ID.

Note: If *return_code* is not set to CM_OK, *security_user_ID* is undefined.

security_user_ID_length *(output)*
Specifies the variable containing the length of the user ID.

Note: If *return_code* is not set to CM_OK, *security_user_ID_length* is undefined.

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_PROGRAM_STATE_CHECK
  This value indicates that the program is in **Initialize-Incoming** state.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

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

Related Information

"Set_Conversation_Security_User_ID (CMSCSU)" on page 153 discusses the setting of the *security_user_ID* characteristic.

"Automatic Conversion of Characteristics" on page 22 provides further information on the automatic conversion of the *security_user_ID* parameter.

Extract_Sync_Level (CMESL)

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 resource recovery processing, this value specifies that this conversation is a protected resource. The programs can issue resource recovery calls and will recognize returned parameters relating to resource recovery operations. The programs can also perform confirmation processing.

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_PROGRAM_STATE_CHECK
  This value indicates that the *conversation_ID* specifies a conversation in **Initialize-Incoming** state.
- CM_OPERATION_NOT_ACCEPTED
- 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 180 provides more information on the *sync_level* characteristic.

Extract_TP_Name (CMETPN)

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

For a conversation established using Initialize_Conversation, *TP_name* is the value set from the side information referenced by *sym_dest_name* or set by the Set_TP_Name call.

For a conversation established by Accept_Conversation or Accept_Incoming, *TP_name* is the value included in the conversation startup request. Since this value comes from the *TP_name* characteristic of the remote program, the values are the same at both ends of a conversation.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

TP_name *(output)*
Specifies the variable containing the *TP_name* for the specified conversation.

Note: Unless *return_code* is set to CM_OK, the value in *TP_name* is not meaningful.

TP_name_length *(output)*
Specifies the variable containing the length of the returned *TP_name*.

Note: Unless *return_code* is set to CM_OK, the value in *TP_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_PROGRAM_STATE_CHECK
  This value indicates that the conversation is in **Initialize-Incoming** state.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

This call is used by programs that accept multiple conversations. Extract_TP_Name allows the program to determine which local name was specified in the incoming conversation startup request.

Related Information

"Specify_Local_TP_Name (CMSLTP)" on page 184 provides more information on handling multiple names within a single program.

Flush (CMFLUS)

A program uses the Flush (CMFLUS) call to empty the local system's send buffer for a given conversation. When notified by CPI Communications that a Flush has been issued, the system sends any information it has buffered to the remote system. 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_OPERATION_INCOMPLETE
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The conversation is not in **Send**, **Send-Pending**, or **Defer-Receive** state.
  - For a conversation with *sync_level* set to CM_SYNC_POINT, the conversation's context is in the **Backout-Required** condition. The Flush call is not allowed for this conversation while its context is in this condition.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation ID.
- CM_OPERATION_NOT_ACCEPTED
- 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 system normally buffers the data from consecutive Send_Data calls until it has a sufficient amount for transmission. Only then does the local system transmit the buffered data.

   The local program can issue a Flush call to cause the system 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 system to flush its send buffer only when the system has some information to transmit. If the system has no information in its send buffer, nothing is transmitted to the remote system.

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

"Data Buffering and Transmission" on page 43 discusses the conditions for data transmission.

"Example 4: Validation and Confirmation of Data Reception" on page 46 shows an example of how a program can use the Flush call to establish a conversation immediately.

"Allocate (CMALLC)" on page 77 provides more information on how information is buffered from the Allocate call.

"Send_Data (CMSEND)" on page 137 provides more information on how information is buffered from the Send_Data call.

"Send_Error (CMSERR)" on page 143 provides more information on how information is buffered from the Send_Error call.

"Set_Send_Type (CMSST)" on page 178 discusses alternative methods of achieving the Flush function.

Initialize_Conversation (CMINIT)

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 destination name. 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.

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.

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 20.
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 or Initialize_For_Incoming, 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 15.
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 program may obtain information about its partner program (for example, *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 (using Set_Partner_LU_Name, Set_TP_Name, and Set_Mode_Name calls) before issuing the Allocate call.

Related Information

"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 37 shows an example program flow where Initialize_Conversation is used.

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

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

Initialize_For_Incoming (CMINIC)

Initialize_For_Incoming (CMINIC) is used by a program to initialize values for various conversation characteristics before the conversation is accepted with an Accept_Incoming call.

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

Format

```
CALL CMINIC(conversation_ID,
            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_For_Incoming 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.

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

State Changes

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

Usage Notes

1. For a list of the conversation characteristics initialized when the Initialize_For_Incoming call completes successfully, see Table 3 on page 20.
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.
3. The call is designed for use with Accept_Incoming. As shown in Table 3, when Initialize_For_Incoming completes, certain conversation characteristics are initialized. When the Accept_Incoming call completes, the remaining applicable conversation characteristics are initialized.
4. The Initialize_For_Incoming and Accept_Incoming calls can be used by a program to accept multiple conversations.

Related Information

"Accept_Incoming (CMACCI)" on page 75 describes how the *conversation_ID* is used when an incoming conversation is accepted by a program.

"Conversation Characteristics" on page 18 describes how the conversation characteristics are initialized by the Initialize_For_Incoming and Accept_Incoming calls.

"Example 11: Accepting Multiple Conversations Using Blocking Calls" on page 60 and "Example 12: Accepting Multiple Conversations Using Non-Blocking Calls" on page 62 show example program flows where Initialize_For_Incoming is used.

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

Prepare_To_Receive (CMPTR)

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 resource recovery 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_OPERATION_INCOMPLETE
- 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 conversation's context is in the **Backout-Required** condition. The Prepare_To_Receive call is not allowed while the conversation's context is in this condition.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_OPERATION_NOT_ACCEPTED
- 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_OPERATION_INCOMPLETE
- 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 (basic conversations only)
- CM_DEALLOCATED_ABEND_TIMER (basic conversations only)
- CM_SVC_ERROR_PURGING (basic conversations only)
- 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 conversation's context is in the **Backout-Required** condition.
- CM_PROGRAM_PARAMETER_CHECK
  This return code indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_OPERATION_NOT_ACCEPTED
- 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 (basic conversations only)
  - CM_DEALLOCATED_ABEND_TIMER_BO (basic conversations only)
  - 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_OPERATION_INCOMPLETE
- 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 conversation's context is in the **Backout-Required** condition.
- CM_PROGRAM_PARAMETER_CHECK
  This return code indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

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 system'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 system'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 resource recovery commit call, the system'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 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 a resource recovery 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 system buffers the send notification to be sent to the remote partner until the local program issues a call that causes the system to flush its send buffer.

Related Information

"Set_Prepare_To_Receive_Type (CMSPTR)" on page 170 provides more information on the *prepare_to_receive_type* characteristic.

"Set_Sync_Level (CMSSL)" on page 180 discusses the *sync_level* characteristic and its possible values.

"Example 3: The Sending Program Changes the Data Flow Direction" on page 44 and "Example 5: The Receiving Program Changes the Data Flow Direction" on page 48 show example program flows where the Prepare_To_Receive call is used.

Receive (CMRCV)

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 resource recovery 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 the maximum buffer size supported by the system. The maximum buffer size is at least 32767 bytes. See Usage Note 11 on page 131 for additional information about determining the maximum buffer size.

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 conversations 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 does not receive data on this call, the value contained in *received_length* is meaningless.

Note: Data is received only if *return_code* is set to CM_OK or CM_DEALLOCATED_NORMAL, and *data_received* is not set to CM_NO_DATA_RECEIVED.

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 a resource recovery 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 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 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 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 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 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 135 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_OPERATION_INCOMPLETE
- 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 (basic conversations only)
- CM_DEALLOCATED_ABEND_TIMER (basic conversations only)
- CM_SVC_ERROR_NO_TRUNC (basic conversations only)
- CM_SVC_ERROR_PURGING (basic conversations only)
- CM_OPERATION_NOT_ACCEPTED
- 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 (basic conversations only)
  - CM_DEALLOCATED_ABEND_TIMER_BO (basic conversations only)
  - 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_OPERATION_INCOMPLETE
- 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 (basic conversations only)
- CM_DEALLOCATED_ABEND_TIMER (basic conversations only)

- CM_SVC_ERROR_NO_TRUNC (basic conversations only)
- CM_SVC_ERROR_PURGING (basic conversations only)
- CM_OPERATION_NOT_ACCEPTED
- 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 (basic conversations only)
  - CM_DEALLOCATED_ABEND_TIMER_BO (basic conversations only)
  - 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_OPERATION_INCOMPLETE
  This value is only received when *receive_type* is set to CM_RECEIVE_AND_WAIT.
- 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 conversations only)
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_DEALLOCATED_ABEND_SVC (basic conversations only)
- CM_DEALLOCATED_ABEND_TIMER (basic conversations only)
- CM_SVC_ERROR_NO_TRUNC (basic conversations only)
- CM_SVC_ERROR_PURGING (basic conversations only)
- CM_SVC_ERROR_TRUNC (basic conversations only)
- CM_UNSUCCESSFUL
  This value indicates that *receive_type* is set to CM_RECEIVE_IMMEDIATE, but there is no data or status to receive.
- CM_OPERATION_NOT_ACCEPTED
- 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 (basic conversations only)
  - CM_DEALLOCATED_ABEND_TIMER_BO (basic conversations only)
  - 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.
  - For a conversation with *sync_level* set to CM_SYNC_POINT, the conversation's context is in the **Backout-Required** condition. The Receive call is not allowed for this conversation while its context 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 that exceeds the range permitted by the implementation. The maximum value of the length in each implementation is at least 32767. See Usage Note 11 on page 131 for additional information about determining the maximum buffer size.

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, or CM_COMPLETE_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 information 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 system will flush its send buffer and send all buffered information to the remote program. The local system 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 system 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 119 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 15.
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 either:
     – An indication of a change in the state of the conversation—that is, a change to **Send**, **Send-Pending**, **Confirm**, **Confirm-Send**, **Confirm-Deallocate**, **Sync-Point**, **Sync-Point-Send**, **Sync-Point-Deallocate**, or **Reset** state
     – An error indication, such as a CM_PROGRAM_ERROR_NO_TRUNC return code.
   - *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 *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 the above cases, the program receives no data.

If the conversation is mapped and a null data record is available (resulting from a Send_Data call with *send_length* set to 0), the *data_received* parameter is set to CM_COMPLETE_DATA_RECEIVED and the *received_length* parameter is set to 0.

Note: When *requested_length* is set to zero, receipt of either data or status can be indicated, but not both. The only exception to this rule is when a null data record is available for receipt on a mapped conversation. In that case, receipt of the null data record and status can both be indicated.

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.
11. A program must not specify a value in the *requested_length* parameter that is greater than the maximum the implementation can support. The maximum may vary from system to system. The program can use the Extract_Maximum_Buffer_Size call to determine the maximum supported by the local system. The program can achieve portability across different systems by using one of the following methods:
    a. Never using a *requested_length* value greater than 32767.
    b. Using the Extract_Maximum_Buffer_Size call to determine the maximum buffer size supported by the system and never setting *requested_length* greater than that maximum buffer size.

    The program should also be aware that the CM_INCOMPLETE_DATA_RECEIVED value of the *data_received* parameter may be returned when the maximum buffer size differs across the systems.
12. When *processing_mode* is set to CM_NON_BLOCKING and *receive_type* is set to CM_RECEIVE_IMMEDIATE, the call completes immediately. If information is not available, *return_code* is set to CM_UNSUCCESSFUL.

Related Information

"Send_Data (CMSEND)" on page 137 provides more information on complete and incomplete logical records and data records.

"Program Partners and Conversations" on page 9 and "Set_Fill (CMSF)" on page 162 further discuss 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 135 discusses how a program can place its end of the conversation into **Receive** state.

“Set_Receive_Type (CMSRT)” on page 174 discusses the *receive_type* characteristic and its various values.

Release_Local_TP_Name (CMRLTP)

Release_Local_TP_Name (CMRLTP) is used by a program to release a name. The name is no longer associated with the program.

Format

```
CALL CMRLTP(TP_name,
            TP_name_length,
            return_code)
```

Parameters

TP_name *(input)*
Specifies the name to be released.

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* variable can have one of the following values:

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *TP_name* specifies a name that is not associated with this program.
  - 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 any state changes.

Usage Notes

1. If a *return_code* other than CM_OK is returned on the call, the names associated with the current program remain unchanged.
2. The names used to satisfy an outstanding Accept_Incoming or Accept_Conversation call are not changed by the Release_Local_TP_Name call. The released name will not be used to satisfy future Accept_Incoming or Accept_Conversation calls.
3. A TP can release a name that was taken from the conversation startup request and used to start the program.
4. If a TP has released all names, no incoming conversations can be accepted. Subsequent Accept_Incoming and Accept_Conversation calls will receive the CM_PROGRAM_STATE_CHECK return code.

Related Information

"Specify_Local_TP_Name (CMSLTP)" on page 184 describes how local names are associated with a program.

Request_To_Send (CMRTS)

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_OPERATION_INCOMPLETE
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - 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 conversation's context is in the **Backout-Required** condition. The Request_To_Send call is not allowed for this conversation while its context is in this condition.
- CM_PROGRAM_PARAMETER_CHECK
  This return code indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_OPERATION_NOT_ACCEPTED
- 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 123 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, Send_Error, Confirm, or Test_Request_To_Send_Received call). However, the value can also be returned on a Receive call.
3. When the remote system 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 system 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 123 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 48 shows an example program flow using the Request_To_Send call.

Send_Data (CMSEND)

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 system 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 system. It is used, for example, by the system'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 the maximum buffer size supported by the system. The maximum buffer size is at least 32767 bytes. See Usage Note 9 on page 142 for additional information about determining the maximum buffer size. The *send_length* parameter 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_OPERATION_INCOMPLETE
- 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 (basic conversations only)
- CM_DEALLOCATED_ABEND_TIMER (basic conversations only)
- CM_SVC_ERROR_PURGING (basic conversations only)
- 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.
  - For a conversation with *sync_level* set to CM_SYNC_POINT, the conversation's context is in the **Backout-Required** condition. The Send_Data call is not allowed for this conversation while its context 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* exceeds the range permitted by the implementation. The maximum value of the length in each implementation is at least 32767. See Usage Note 9 on page 142 for additional information about determining the maximum buffer size.
  - 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_OPERATION_NOT_ACCEPTED
- 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 (basic conversations only)
  - CM_DEALLOCATED_ABEND_TIMER_BO (basic conversations only)
  - 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 system buffers the data to be sent to the remote system 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 system 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 135 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 system. 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
   - Resource recovery 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 system to flush its send buffer, which includes sending the truncated record. The system 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. If the *log_data* characteristic is not null and these conditions occur, log data is sent.
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.

9. A program must not specify a value in the *send_length* parameter that is greater than the maximum the implementation can support. The maximum may vary from system to system. The program can use the Extract_Maximum_Buffer_Size call to find out the maximum buffer size supported by the local system. The program can achieve portability across different systems by using one of the following methods:
   a. Never using a *send_length* value greater than 32767.
   b. Using the Extract_Maximum_Buffer_Size call to determine the maximum buffer size supported by the system and never setting *send_length* greater than that maximum buffer size.

Related Information

"Receive (CMRCV)" on page 123 provides more information on the *data_received* parameter.

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

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

"Data Buffering and Transmission" on page 43 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 178 provides more information on the *send_type* conversation characteristic and the use of it in combination with calls to Send_Data.

Send_Error (CMSERR)

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 system 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_OPERATION_INCOMPLETE
- 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 (basic conversations only)
- CM_DEALLOCATED_ABEND_TIMER (basic conversations only)
- CM_SVC_ERROR_PURGING (basic conversations only)
- CM_PROGRAM_PARAMETER_CHECK
  The *conversation_ID* specifies an unassigned conversation identifier.
- CM_OPERATION_NOT_ACCEPTED
- 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 (basic conversations only)
  - CM_DEALLOCATED_ABEND_TIMER_BO (basic conversations only)
  - CM_PROGRAM_STATE_CHECK
    This return code indicates that for a conversation with *sync_level* set to CM_SYNC_POINT, the conversation's context is in the **Backout-Required** condition. The Send_Error call is not allowed for this conversation while its context 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_OPERATION_INCOMPLETE
- CM_DEALLOCATED_NORMAL
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_PROGRAM_PARAMETER_CHECK
  The *conversation_ID* specifies an unassigned conversation identifier.
- CM_OPERATION_NOT_ACCEPTED
- 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 for a conversation with *sync_level* set to CM_SYNC_POINT, the conversation's context is in the

Backout-Required condition. The Send_Error call is not allowed for this conversation while its context 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_OPERATION_INCOMPLETE
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_PROGRAM_PARAMETER_CHECK
  The *conversation_ID* specifies an unassigned conversation identifier.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR
- These values are returned only when *sync_level* is set to CM_SYNC_POINT:
  - CM_TAKE_BACKOUT
  - CM_PROGRAM_STATE_CHECK
    This return code indicates that for a conversation with *sync_level* set to CM_SYNC_POINT, the conversation's context is in the **Backout-Required** condition. The Send_Error call is not allowed for this conversation while its context 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_OPERATION_INCOMPLETE
- CM_PROGRAM_PARAMETER_CHECK
  The *conversation_ID* specifies an unassigned conversation identifier.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR
- CM_PROGRAM_STATE_CHECK
  This return code indicates that for a conversation with *sync_level* set to CM_SYNC_POINT, the conversation's context is in the **Backout-Required** condition. The Send_Error call is not allowed for this conversation while its context 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**, **Defer-Deallocate**, or **Initialize-Incoming** state and *return_code* has one of the following values:

- CM_PROGRAM_PARAMETER_CHECK
  The *conversation_ID* specifies an unassigned identifier.
- CM_OPERATION_NOT_ACCEPTED
- 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 system can send the error notification to the remote system immediately (during the processing of this call), or the system can delay sending the notification until a later time. If the system 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 system 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 137). 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 (basic conversations only)
   - CM_DEALLOCATED_ABEND_TIMER (basic conversations only)

   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 (basic conversations only)
   - CM_DEALLOCATED_ABEND_TIMER_BO (basic conversations only)

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

   When the return code CM_TAKE_BACKOUT is purged, the remote system 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.
   - Resource recovery 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 system 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.

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

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

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

   Because the system 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 50 and "Example 7: Error Direction and Send-Pending State" on page 52 provide example program flows using Send_Error and the **Send-Pending** state; "Set_Error_Direction (CMSED)" on page 160 provides further information on the *error_direction* characteristic.

"Usage Notes" of "Request_To_Send (CMRTS)" on page 135 provides more information on how a conversation enters **Receive** state.

"Send_Data (CMSEND)" on page 137 discusses basic conversations and logical records.

"Set_Log_Data (CMSLD)" on page 164 provides a description of the *log_data* characteristic.

Set_Conversation_Security_Password (CMSCSP)

Set_Conversation_Security_Password (CMSCSP) is used by a program to set the *security_password* and *security_password_length* characteristics for a conversation. Set_Conversation_Security_Password overrides the current values, which were originally acquired from the side information using *sym_dest_name*.

This call does not change the values in the side information. It only changes the *security_password* and *security_password_length* characteristics for this conversation.

Note: A program cannot issue the Set_Conversation_Security_Password call after an Allocate call is issued. Only the program that initiates the conversation (issues the Initialize_Conversation call) can issue Set_Conversation_Security_Password. A program can only specify a password when *conversation_security_type* is set to CM_SECURITY_PROGRAM.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

security_password *(input)*
Specifies the password to be included in the conversation startup request. The partner system uses this value and the user ID to validate the user's access to the remote program. The password is stored temporarily by node services, and is erased at the successful completion of an Allocate call.

security_password_length *(input)*
Specifies the length of the password. The length can be from 0 to 10 bytes. If zero, the *security_password_length* characteristic is set to zero (effectively setting the *security_password* characteristic to the null string), and the *security_password* 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_STATE_CHECK
  This value indicates one of the following:
  - The conversation is not in **Initialize** state.
  - *conversation_security_type* is not set to CM_SECURITY_PROGRAM.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *security_password_length* is less than 0 or greater than 10.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

1. When a program issues Set_Conversation_Security_Password, a user ID must also be supplied. The user ID comes from side information or is set by the program issuing Set_Conversation_Security_User_ID.
2. Specify *security_password* using the local system's native encoding. CPI Communications automatically converts the *security_password* from the native encoding where necessary.
3. Specification of a password that is not valid is not detected on this call. It is detected by the partner system when it receives the conversation startup request. The partner system returns an error indication to the local system, which reports the error to the program by means of the CM_SECURITY_NOT_VALID return code on a call subsequent to the Allocate call.
4. If a *return_code* other than CM_OK is returned on the call, the *security_password* and *security_password_length* characteristics are unchanged.

Related Information

"Set_Conversation_Security_Type (CMSCST)" on page 151 provides more information on the *conversation_security_type* characteristic.

"Set_Conversation_Security_User_ID (CMSCSU)" on page 153 provides more information on the *security_user_ID* characteristic.

"Automatic Conversion of Characteristics" on page 22 provides further information on the automatic conversion of the *security_password* parameter.

"Conversation Security" on page 24 provides further information on security.

Set_Conversation_Security_Type (CMSCST)

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

This call does not change the value in the side information. It only changes the *conversation_security_type* characteristic for this conversation.

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

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

conversation_security_type *(input)*
Specifies the type of access security information to be sent in the conversation startup request to the partner system. The access security information, if present, consists of either a user ID, or a user ID and a password. It is used by the partner system to validate the user's access to the remote program.

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

- CM_SECURITY_NONE
  No access security information is included in the conversation startup request.
- CM_SECURITY_SAME
  The security parameters maintained by node services for the program's current context when the program issues the Allocate call are used to set the access security information included in the conversation startup request.
- CM_SECURITY_PROGRAM
  The values of the *security_user_ID* and *security_password* characteristics are used to set the access security information included in the conversation startup request.

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 return code indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *conversation_security_type* specifies an undefined value.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

1. A *conversation_security_type* of CM_SECURITY_SAME is intended for use between nodes which have the same set of user IDs and which accept user validation performed on one node as verifying the user for all nodes. A password is not used in this case.
2. A *conversation_security_type* of CM_SECURITY_PROGRAM requires that a user ID and password be supplied for inclusion in the conversation startup request. These may come from side information or be set by the program using the Set_Conversation_Security_User_ID and Set_Conversation_Security_Password calls.
3. If a *return_code* other than CM_OK is returned on the call, the *conversation_security_type* is unchanged.

Related Information

"Set_Conversation_Security_Password (CMSCSP)" on page 149 discusses setting the *security_password* characteristic.

"Set_Conversation_Security_User_ID (CMSCSU)" on page 153 discusses setting the *security_user_ID* characteristic.

"Conversation Security" on page 24 provides further information on security.

Set_Conversation_Security_User_ID (CMSCSU)

Set_Conversation_Security_User_ID (CMSCSU) is used by a program to set the *security_user_ID* and *security_user_ID_length* characteristics for a conversation. Set_Conversation_Security_User_ID overrides the current values, which were originally acquired from the side information using *sym_dest_name*.

This call does not change the values in the side information. It only changes the *security_user_ID* and *security_user_ID_length* characteristics for this conversation.

Note: A program cannot issue the Set_Conversation_Security_User_ID call after an Allocate call is issued. Only the program that initiates the conversation (issues the Initialize_Conversation call) can issue Set_Conversation_Security_User_ID. A program can only specify an access security user ID when *conversation_security_type* is set to CM_SECURITY_PROGRAM.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

security_user_ID *(input)*
Specifies the user ID to be included in the conversation startup request. The partner system uses this value and the password to validate the user's access to the remote program. In addition, the partner system may use the user ID for auditing or accounting purposes.

security_user_ID_length *(input)*
Specifies the length of the user ID. The length can be from 0 to 10 bytes. If zero, the *security_password_length* characteristic is set to zero (effectively setting the *security_password* characteristic to the null string), and the *security_password* 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_STATE_CHECK
  This value indicates one of the following:
  - The conversation is not in **Initialize** state.
  - The *conversation_security_type* is not set to CM_SECURITY_PROGRAM.
- CM_PROGRAM_PARAMETER_CHECK
  This return code indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *security_user_ID_length* is less than 0 or greater than 10.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

1. When a program issues Set_Conversation_Security_User_ID, a password must also be supplied. The password comes from side information or is set by the program using Set_Conversation_Security_Password.
2. Specify *security_user_ID* using the local system's native encoding. CPI Communications automatically converts the *security_user_ID* from the native encoding where necessary.
3. Specification of a security user ID that is not valid is not detected on this call. It is detected by the partner system when it receives the conversation startup request. The partner system returns an error indication to the local system, which reports the error to the program by means of the CM_SECURITY_NOT_VALID return code on a call subsequent to the Allocate call.
4. If a *return_code* other than CM_OK is returned on the call, the *security_user_ID* and *security_user_ID_length* characteristics are unchanged.

Related Information

"Set_Conversation_Security_Password (CMSCSP)" on page 149 provides more information on the *security_password* characteristic.

"Set_Conversation_Security_Type (CMSCST)" on page 151 provides more information on the *conversation_security_type* characteristic.

"Automatic Conversion of Characteristics" on page 22 provides further information on the automatic conversion of the *security_user_ID* parameter.

"Conversation Security" on page 24 provides further information on security.

Set_Conversation_Type (CMSCT)

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_Type 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_OPERATION_NOT_ACCEPTED
- 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 9 and the "Usage Notes" section of "Send_Data (CMSEND)" on page 137 provide more information on the differences between mapped and basic conversations.

Set_Deallocate_Type (CMSDT)

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 the Initialize_Conversation, Accept_Conversation, or Initialize_For_Incoming 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 a resource recovery 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 backout call instead of a commit, the conversation is not deallocated. See "Deallocate (CMDEAL)" on page 91 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_OPERATION_NOT_ACCEPTED
- 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 Commit call. If the Commit call is successful, the conversation is deallocated normally. If the Commit is not successful or if the program issues an SAA resource recovery 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 91 provides further discussion on the use of the *deallocate_type* characteristic in the deallocation of a conversation.

"Set_Sync_Level (CMSSL)" on page 180 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)

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, the Accept_Conversation, or the Initialize_For_Incoming 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_OPERATION_NOT_ACCEPTED
- 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 system 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 52 provides an example program using Set_Error_Direction.

"Send_Error (CMSERR)" on page 143 provides more information on reporting errors.

"Send-Pending State and the error_direction Characteristic" on page 226 provides more information on the *error_direction* characteristic.

Set_Fill (CMSF)

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, Accept_Conversation, or Initialize_For_Incoming call.

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_OPERATION_NOT_ACCEPTED
- 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 123 provides more information on how the *fill* characteristic is used for basic conversations.

Set_Log_Data (CMSLD)

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, Accept_Conversation, or Initialize_For_Incoming call.

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

Format

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

Parameters

conversation_ID *(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_OPERATION_NOT_ACCEPTED
- 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 system 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 system 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 system only once even though an error indication may be issued several times. See usage note 1 on page 164 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 143 and "Deallocate (CMDEAL)" on page 91 provide further discussion on how the *log_data* characteristic is used.

"Automatic Conversion of Characteristics" on page 22 provides further information on the automatic conversion of the *log_data* parameter.

Set_Mode_Name (CMSMN)

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 may require special authority to specify some mode names. For example, SNASVCMG requires special authority with LU 6.2.

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_OPERATION_NOT_ACCEPTED
- 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 system 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 227 discusses SNA service transaction programs.

"Side Information" on page 11 further discusses the *mode_name* conversation characteristic.

"Automatic Conversion of Characteristics" on page 22 provides further information on the automatic conversion of the *mode_name* parameter.

Set_Partner_LU_Name (CMSPLN)

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 system 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_OPERATION_NOT_ACCEPTED
- 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 11 and note 2 of Table 7 on page 198 provide further discussion of the *partner_LU_name* conversation characteristic.

"Automatic Conversion of Characteristics" on page 22 provides further information on the automatic conversion of the *partner_LU_name* parameter.

Set_Prepare_To_Receive_Type (CMSPTR)

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, Accept_Conversation, or Initialize_For_Incoming 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 a resource recovery 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_OPERATION_NOT_ACCEPTED
- 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 119 discusses how the *prepare_to_receive_type* is used.

"Example 5: The Receiving Program Changes the Data Flow Direction" on page 48 shows an example program using the Prepare_To_Receive call.

Set_Processing_Mode (CMSPM)

A program uses the Set_Processing_Mode (CMSPM) call to set the *processing_mode* characteristic of a conversation. The *processing_mode* characteristic indicates whether CPI Communications calls on the specified conversation are to be processed in blocking or non-blocking mode. Set_Processing_Mode overrides the default value of CM_BLOCKING that was assigned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call. The processing mode of a conversation cannot be changed prior to the completion of the previous operation on that conversation.

Format

```
CALL CMSPM(conversation_ID,
           processing_mode,
           return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

processing_mode *(input)*
Specifies the processing mode to be used for this conversation. *processing_mode* can have one of the following values:

- CM_BLOCKING
  Specifies that calls will be processed in blocking mode. Calls complete before control is returned to the program. The CM_OPERATION_INCOMPLETE return code will not be returned on this conversation.
- CM_NON_BLOCKING
  Specifies that calls will be processed in non-blocking mode. If possible, the calls complete immediately. When a call operation cannot complete immediately, CPI Communications returns control to the program with the CM_OPERATION_INCOMPLETE return code. The operation proceeds without blocking the 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 *processing_mode* specifies an undefined value.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

1. If a *return_code* other than CM_OK is returned on the call, the *processing_mode* conversation characteristic is unchanged.
2. When CM_OPERATION_INCOMPLETE is returned from an Accept_Incoming, Allocate, Confirm, Confirmed, Deallocate, Flush, Prepare_To_Receive, Request_To_Send, Send_Data, or Send_Error call, the call operation has not completed. The operation proceeds without blocking the program. The data and buffer areas used in the call are in an indeterminate state and should not be referenced until the operation is completed. The Wait_For_Conversation call is used to determine when an operation is completed. Each call to Wait_For_Conversation will return the conversation identifier and return code (the *conversation_return_code* value) of a completed operation. It is the responsibility of the program to keep track of the operation being performed by each conversation in order to be able to properly interpret the *conversation_return_code* value.
3. Not all languages support the use of non-blocking operations. See "Programming Language Considerations" on page 67 for language restrictions on the use of non-blocking operations.

Related Information

"Non-Blocking Operations" on page 22 discusses the use of non-blocking operations.

"Wait_For_Conversation (CMWAIT)" on page 187 describes the use of Wait_For_Conversation to wait for completion of an outstanding operation.

Set_Receive_Type (CMSRT)

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, Accept_Conversation, or Initialize_For_Incoming call.

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_OPERATION_NOT_ACCEPTED
- 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 123 discusses how the *receive_type* characteristic is used.

"Example 3: The Sending Program Changes the Data Flow Direction" on page 44 discusses how a program can use Set_Receive_Type with a value of CM_RECEIVE_IMMEDIATE.

Set_Return_Control (CMSRC)

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 system 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_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. An allocation error resulting from the local system's failure to obtain a session for the conversation is reported on the Allocate call. An allocation error resulting from the remote system's rejection of the allocation request is reported on a subsequent conversation call.
2. Two systems 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 system the contention winner of the session and the other system the contention loser of the session. The contention-winner system allocates a conversation on a session without asking permission from the contention-loser system. Conversely, the contention-loser system requests permission from the contention-winner system to allocate a conversation on the session, and the contention-winner system either grants or rejects the request. For more 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 77 provides more discussion on the use of the *return_control* characteristic in allocating a conversation.

Set_Send_Type (CMSST)

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, Accept_Conversation, or Initialize_For_Incoming 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_OPERATION_NOT_ACCEPTED
- 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 137 discusses 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 113
- "Confirm (CMCFM)" on page 82
- "Prepare_To_Receive (CMPTR)" on page 119
- "Deallocate (CMDEAL)" on page 91

"Example 5: The Receiving Program Changes the Data Flow Direction" on page 48 shows an example program flow using the Set_Send_Type call.

Set_Sync_Level (CMSSL)

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 the calls of a resource recovery interface). 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 resource recovery interface calls and will recognize returned parameters relating to resource recovery processing. The programs can also perform confirmation processing.

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 a resource recovery interface.
  - The *sync_level* is set to CM_SYNC_POINT and *deallocate_type* is set to CM_FLUSH or CM_CONFIRM.
- CM_OPERATION_NOT_ACCEPTED
- 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 82 and "Confirmed (CMCFMD)" on page 85 provide further information on confirmation processing.

"Example 4: Validation and Confirmation of Data Reception" on page 46 and "Example 8: Sending Program Issues a Commit" on page 54 show how to use the Set_Sync_Level call and how to perform confirm and sync point processing.

Set_TP_Name (CMSTPN)

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

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

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

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

TP_name *(input)*
Specifies the name of the remote program.

Note: A program may require special authority to specify some TP names. For example, SNA service transaction programs require special authority with LU 6.2. (For more information, see "SNA Service Transaction Programs" on page 227.)

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_OPERATION_NOT_ACCEPTED
- 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 227 for more information on privilege and service transaction programs.

“Side Information” on page 11 and notes 3 on page 199 and 6 on page 200 of Table 7 on page 198 provide further discussion of the *TP_name* conversation characteristic.

“Automatic Conversion of Characteristics” on page 22 provides further information on the automatic conversion of the *TP_name* parameter.

Specify_Local_TP_Name (CMSLTP)

A program uses the Specify_Local_TP_Name (CMSLTP) call to associate a name with itself, thus notifying CPI Communications that it can accept conversations destined for the name. A program may have many local names. It can extract the *TP_name* for a particular conversation using the Extract_TP_Name call.

Format

```
CALL CMSLTP(TP_name,
            TP_name_length,
            return_code)
```

Parameters

TP_name *(input)*
Specifies a name to be associated with this 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* variable can have one of the following values:

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *TP_name* specifies a name that is restricted in some way by node services.
  - The *TP_name* has incorrect internal syntax as defined by node services.
  - 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 any state changes.

Usage Notes

1. If a *return_code* other than CM_OK is returned on the call, the names associated with the current program remain unchanged.
2. If the program has an outstanding Accept_Incoming or Accept_Conversation call when it issues Specify_Local_TP_Name, the names used to satisfy the outstanding Accept_Incoming or Accept_Conversation are not affected. The newly specified name will be used for the next Accept_Incoming or Accept_Conversation call.

Related Information

"Accept_Conversation (CMACCP)" on page 73 and "Accept_Incoming (CMACCI)" on page 75 describe how an incoming conversation is accepted by a program.

"Release_Local_TP_Name (CMRLTP)" on page 133 explains additional timing considerations for names associated with the program.

Test_Request_To_Send_Received (CMTRTS)

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 conversation's context is in the **Backout-Required** condition. The Test_Request_To_Send_Received call is not allowed for this conversation while its context is in this condition.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. When the local system 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 through the *request_to_send_received* parameter, the local system discards the notification.

Related Information

"Request_To_Send (CMRTS)" on page 135 provides further discussion of the request-to-send function.

Wait_For_Conversation (CMWAIT)

A program must use the Wait_For_Conversation (CMWAIT) call to wait for the completion of an outstanding operation. An outstanding operation is indicated when the CM_OPERATION_INCOMPLETE *return_code* value is returned on an Accept_Incoming, Allocate, Confirm, Confirmed, Deallocate, Flush, Prepare_To_Receive, Receive, Request_To_Send, Send_Data, or Send_Error call. This can occur when the *processing_mode* conversation characteristic is set to CM_NON_BLOCKING and the requested operation cannot complete immediately.

Format

```
CALL CMWAIT(conversation_ID,
            conversation_return_code,
            return_code)
```

Parameters

conversation_ID *(output)*

Specifies the variable containing the conversation identifier for the completed operation.

Note: Unless *return_code* is set to CM_OK, the value contained in *conversation_ID* is not meaningful.

conversation_return_code *(output)*

Specifies the variable containing the return code for the completed operation. The meaning of this return code depends upon the operation that was started. *conversation_return_code* can have one of the following values:

- CM_OK
- CM_ALLOCATE_FAILURE_NO_RETRY
- CM_ALLOCATE_FAILURE_RETRY
- CM_CONVERSATION_TYPE_MISMATCH
- CM_PIP_NOT_SPECIFIED_CORRECTLY
- CM_SECURITY_NOT_VALID
- CM_SYNC_LVL_NOT_SUPPORTED_LU
- 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_PRODUCT_SPECIFIC_ERROR
- CM_PROGRAM_ERROR_NO_TRUNC
- CM_PROGRAM_ERROR_PURGING
- CM_PROGRAM_ERROR_TRUNC
- 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
- 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
- CM_DEALLOCATED_NORMAL_BO

Note: Unless *return_code* is set to CM_OK, the value contained in *conversation_return_code* is not meaningful.

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

- CM_OK
  This value indicates that an outstanding operation has completed and that the *conversation_ID* and *conversation_return_code* have been returned.
- CM_SYSTEM_EVENT
  This value indicates that, rather than an outstanding operation on a conversation, an event (such as a signal) recognized by the program has occurred. The Wait_For_Conversation call returns this return code value to allow the program to decide whether to reissue the Wait_For_Conversation or to perform other processing.
- CM_PROGRAM_STATE_CHECK
  This value indicates that there were no outstanding operations for the program.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

When *return_code* is set to CM_OK, the conversation identified by *conversation_ID* may change state. The new state is determined by the operation that completed, the return code for that operation (the *conversation_return_code* value), and the other factors that affect state transitions.

Usage Notes

1. Wait_For_Conversation waits for the completion of any outstanding operation on any conversation. It is the responsibility of the program to keep track of the operation in progress on each conversation in order to be able to interpret properly the *conversation_return_code* value.
2. It is the responsibility of the event-handling portion of the program to record sufficient information for the program to decide how to proceed on receipt of the CM_SYSTEM_EVENT return code.
3. The program's current context is not changed as a result of the completion of a Wait_For_Conversation call.

Related Information

"Set_Processing_Mode (CMSPM)" on page 172 describes setting the *processing_mode* conversation characteristic.

"Cancel_Conversation (CMCANC)" on page 80 describes the means for terminating an operation before it is completed.

"Non-Blocking Operations" on page 22 discusses the use of non-blocking operations.

Wait_For_Conversation (CMWAIT)

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 26, 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 192 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 several programming languages in Appendix E, "Pseudonym Files" on page 231. See Appendix F, "Sample Programs" on page 297 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.

© Copyright IBM Corp. 1988, 1993

Variable Definitions

Table 5 (Page 1 of 3). Variables/Characteristics and Their Possible Values

Variable or Characteristic Name	Pseudonym Values	Integer Values
conversation_return_code	See *return_code*.	
conversation_security_type	CM_SECURITY_NONE	0
	CM_SECURITY_SAME	1
	CM_SECURITY_PROGRAM	2
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
	CM_INITIALIZE_INCOMING_STATE	14
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
processing_mode	CM_BLOCKING	0
	CM_NON_BLOCKING	1
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 3). Variables/Characteristics and Their Possible Values

Variable or Characteristic Name	Pseudonym Values	Integer Values
return_code and *conversation_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_OPERATION_INCOMPLETE	35
	CM_SYSTEM_EVENT	36
	CM_OPERATION_NOT_ACCEPTED	37
	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

Table 5 (Page 3 of 3). Variables/Characteristics and Their Possible Values

Variable or Characteristic Name	Pseudonym Values	Integer Values
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 EBCDIC hexadecimal and graphic representations, are provided in Table 6. See the *SNA Formats* manual (GA27-3136) for more information on character sets.

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

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

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

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

Variable Types

CPI Communications makes use of two variable types, integer and character string. Table 7 on page 198 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 195, 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 198).
- **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, as is discussed in note 5 on page 199 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.

Variable Definitions

Table 7 (Page 1 of 2). Variable Types and Lengths

Variable	Variable Type	Character Set	Length
buffer [1, 2]	Character string	no restriction	0-max supported by system
buffer_length [1]	Integer	N/A	32 bits
context_ID	Character string	no restriction	1-32 bytes
context_ID_length	Integer	N/A	32 bits
conversation_ID	Character string	no restriction	8 bytes
conversation_return_code	Integer	N/A	32 bits
conversation_security_type	Integer	N/A	32 bits
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 [3]	Character string	00640	0-512 bytes
log_data_length	Integer	N/A	32 bits
maximum_buffer_size [2]	Integer	N/A	32 bits
mode_name [3, 4, 8]	Character string	01134	0-8 bytes
mode_name_length	Integer	N/A	32 bits
partner_LU_name [3, 4, 5]	Character string	01134	1-17 bytes
partner_LU_name_length	Integer	N/A	32 bits
prepare_to_receive_type	Integer	N/A	32 bits
processing_mode	Integer	N/A	32 bits
receive_type	Integer	N/A	32 bits
received_length [2]	Integer	N/A	32 bits
request_to_send_received	Integer	N/A	32 bits
requested_length [2]	Integer	N/A	32 bits
return_code	Integer	N/A	32 bits
return_control	Integer	N/A	32 bits
security_password [3]	Character string	00640	0-10 bytes
security_password_length	Integer	N/A	32 bits
security_user_ID [3, 4]	Character string	00640	0-10 bytes
security_user_ID_length	Integer	N/A	32 bits
send_length [2]	Integer	N/A	32 bits
send_type	Integer	N/A	32 bits
status_received	Integer	N/A	32 bits
sym_dest_name [3, 7]	Character string	01134	8 bytes
sync_level	Integer	N/A	32 bits

Table 7 (Page 2 of 2). Variable Types and Lengths			
Variable	**Variable Type**	**Character Set**	**Length**
TP_name [3, 6]	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. For character data in character data set 00640, this conversion can be accomplished by Convert_Outgoing in the sending program and by Convert_Incoming in the receiving program. The maximum allowed value of the *buffer_length* parameter on the Convert_Incoming and Convert_Outgoing calls is implementation-specific.
2. The maximum buffer size for sending and receiving data may vary from system to system. The maximum buffer size is at least 32767.
3. 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 22 for more information on automatic conversion of these fields.
4. Because the *mode_name*, *partner_LU_name*, and *security_user_ID* 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 some 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.
5. 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.

6. 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 225 and “SNA Service Transaction Programs” on page 227 for details.
7. 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 115 for more information.
8. The four names in the following list are mode names defined by the LU 6.2 architecture 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 occurred 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 192 of Appendix A, “Variables and Characteristics.”

The valid *return_code* values are described below:

© Copyright IBM Corp. 1988, 1993

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 system or remote system. 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 system 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 system 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 system 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_DEALLOCATE_ABEND, or the remote system 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 conversation's context is in the **Backout-Required** condition and the program must issue a resource recovery backout call in order to restore all of the context's protected resources to their status as of the last synchronization point.

CM_DEALLOCATED_ABEND_SVC
This return code is returned for basic conversations only.

It 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 system 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 basic 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 conversation's context is in the **Backout-Required** condition and the program must issue a resource recovery backout call in order to restore all of the context's protected resources to their status as of the last synchronization point.

CM_DEALLOCATED_ABEND_TIMER
This return code is returned only for basic conversations.

In addition, it 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 basic 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 conversation's context is in the **Backout-Required** condition and the program must issue a resource recovery backout call in order to restore all of the context's 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 system. This purged information may include an abend deallocation notification from the remote program or system. 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 conversation's context is in the **Backout-Required** condition and the program must issue a resource recovery backout call in order to restore all of the context's 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_OPERATION_INCOMPLETE
A non-blocking operation has been started on the conversation but is not complete. This return code is returned when *processing_mode* is set to CM_NON_BLOCKING for the conversation and the call is suspended waiting for incoming data, buffers, or other resources. A program must use the Wait_For_Conversation call to wait for the operation to complete and to retrieve the return code for the completed operation. The state of the conversation remains unchanged.

CM_OPERATION_NOT_ACCEPTED
A previous operation on this conversation is incomplete. This return code is returned when there is an outstanding operation on the conversation, as

indicated by the CM_OPERATION_INCOMPLETE return code to a previous call. On a system that supports multiple program threads, when one thread has started an operation that has not completed, this return code may be returned on a call made by another thread on the same conversation. The state of the conversation remains unchanged.

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 while its end of the conversation was in **Receive** or **Confirm** state. The call may have caused information enroute to the remote program to be purged (discarded), but not necessarily.

  Purging occurs when the remote program issues Send_Error for a conversation in **Receive** state before receiving all the information being sent by the local program. No purging occurs when the remote program issues Send_Error for a conversation in **Receive** state if the remote program has already received all the information sent by the local program. Also, no purging occurs when the remote program issues Send_Error for a conversation in **Confirm** state.

  When information is purged, the purging can occur at the local system, the remote system, or both.

- 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.
- For a conversation with *sync_level* set to CM_SYNC_POINT, the conversation's context is in the **Backout-Required** condition. The call issued is not allowed for this conversation while its context 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 systems. 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 system 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 systems. 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 conversation's context is in the **Backout-Required** condition and the program must issue a resource recovery backout call in order to restore all of the context's 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 conversation's context is in the **Backout-Required** condition and the program must issue a resource recovery backout call in order to restore all of the context's protected resources to their status as of the last synchronization point.

CM_SECURITY_NOT_VALID
: The remote system 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 for basic conversations. In addition, it 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 for basic conversations. In addition, it 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 system 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_SYSTEM_EVENT
The Wait_For_Conversation call was being executed when an event (such as a signal) handled by the program occurred. Wait_For_Conversation returns this return code to allow the program to reissue the Wait_For_Conversation call or to perform other processing. It is the responsibility of the event-handling portion of the program to record sufficient information for the program to decide how to proceed upon receipt of this return code. The state of the conversation remains unchanged.

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 a resource recovery backout call, and the local application must issue a backout call in order to restore all protected resources for a context to their status as of the last synchronization point. The conversation's context is in the **Backout-Required** condition upon receipt of this return code. Once the local program issues a 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 system rejected the allocation request because the local program specified a remote program name that the remote system 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 system rejected the allocation request because the local program specified a remote program that the remote system 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.

Return Codes

CM_TP_NOT_AVAILABLE_RETRY
: The remote system rejected the allocation request because the local program specified a remote program that the remote system 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 25, 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 219 describes the state transitions that are allowed for the CPI Communications calls.

Table 9 on page 223 shows the effects of SAA resource recovery 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.

© Copyright IBM Corp. 1988, 1993

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
oi	CM_OPERATION_INCOMPLETE
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

State Table

Abbreviation	Meaning
rb	rb means one of the following: • CM_RESOURCE_FAIL_NO_RETRY_BO • CM_RESOURCE_FAILURE_RETRY_BO
rf	rf means one of the following: • CM_RESOURCE_FAILURE_NO_RETRY • CM_RESOURCE_FAILURE_RETRY
se	CM_SYSTEM_EVENT
sc	CM_PROGRAM_STATE_CHECK
un	CM_UNSUCCESSFUL

Notes:

1. 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.
2. The CM_OPERATION_NOT_ACCEPTED return code is not included in the state table. A program receives CM_OPERATION_NOT_ACCEPTED when it issues any call (except Cancel_Conversation and Wait_For_Conversation) on a conversation with an outstanding operation, regardless of the state. No conversation state transition occurs.

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
1-14	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 conversation's context 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 conversation's context 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 accepted the conversation goes to **Receive** state.
%	Wait_For_Conversation can only be issued when one or more of the conversations in which a program is participating has received a *return_code* of CM_OPERATION_INCOMPLETE. When Wait_For_Conversation completes with a *return_code* of CM_OK, it indicates a conversation on which an operation has completed. That conversation then moves to the appropriate state as determined by the *conversation_return_code* for the operation that is now completed and by the other factors that determine state transitions.

How to Use the State Table

Each CPI Communications call[2] 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 *processing_mode* set to CM_NON_BLOCKING, the following special considerations apply:

- When a call gets the CM_OPERATION_INCOMPLETE return code, the operation remains in progress as an oustanding operation. The only valid calls on a conversation with an outstanding operation are Wait_For_Conversation and Cancel_Conversation. Any other calls on that conversation get a return code of CM_OPERATION_NOT_ACCEPTED, and no conversation state transition occurs.
- The CM_OPERATION_NOT_ACCEPTED return code is not included in the state table.

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 context 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 context 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 its context 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 223 shows the effects of SAA resource recovery Commit and Backout calls on CPI Communications conversation states.

[2] Only the calls that affect conversation states are included in the State table.

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 several 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 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 213 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.
- There is no state transition for a conversation in **Send** state when a return code of CM_PROGRAM_STATE_CHECK (“sc”) is returned. This return code will never be returned when this call is issued from **Send-Pending** state.

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

Inputs	States 1–8 and 14 are used by all conversations								Used only by conversations with *sync_level* set to CM_SYNC_POINT					
	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	Initialize-Incoming 14
Accept_Conversation	↓	/	/	/	/	/	/	/	/	/	/	/	/	/
[ok]	4													
[sc]	–													
Accept_Incoming	[pc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	↓
[ok]														4
[oi,pc,sc]														–
Allocate	[pc]	↓′	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]		3												
[ae]		1												
[oi,pc,pe,un]		–												
Cancel_Conversation	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok]		1^	1^	1^	1^	1^	1^	1^	1^	1^	1^	1^	1^	1^
[pc]		–	–	–	–	–	–	–	–	–	–	–	–	–
Confirm	[pc]	[sc]	↓′	[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					
[oi,pc]			–		–				–					
[sc]			–		/				/					
Confirmed	[pc]	[sc]	[sc]	[sc]	[sc]	↓′	↓′	↓′	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]						4	3	1						
[oi,pc]						–	–	–						
Deallocate(A)	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok]		1	1^	1^	1^	1^	1^	1^	1^	1^	1^	1^	1^	1
[oi,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–
Deallocate(C)	[pc]	[sc]	↓	[sc]	↓	[sc]	[sc]	[sc]	/	/	/	/	/	[sc]
[ok,da,rf]			1		1									
[ae]			1		/									
[ep]			4		4									
[oi,pc]			–		–									
[sc]			–		/									
Deallocate(F)	[pc]	[sc]	↓	[sc]	↓	[sc]	[sc]	[sc]	/	/	/	/	/	[sc]
[ok]			1		1									
[oi,pc]			–		–									
[sc]			–		/									
Deallocate(S)	[pc]	[sc]	↓′	[sc]	↓′	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]			10		10									
[oi,pc]			–		–									
[sc]			–		/									
Extract_Conv_Context	[pc]	[sc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,sc]			–	–	–	–	–	–	–	–	–	–	–	
Extract_Conv_State	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,bo,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–
Extract_Conv_Type	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	
Extract_Mode_Name	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	
Extract_Part_LU_Name	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	
Extr_Sec_User_ID	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	

State Table

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

Inputs	States 1–8 and 14 are used by all conversations								Used only by conversations with *sync_level* set to CM_SYNC_POINT					
	Reset	**Initialize**	**Send**	**Receive**	**Send-Pending**	**Confirm**	**Confirm-Send**	**Confirm-Deallocate**	**Defer-Receive**	**Defer-Deallocate**	**Sync-Point**	**Sync-Point Send**	**Sync-Point Deallocate**	**Initialize-Incoming**
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
Extract_Sync_Level	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	
Extract_TP_Name	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	
Flush	[pc]	[sc]	↓′	[sc]	↓′	[sc]	[sc]	[sc]	↓′	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]			–		3				4					
[oi,pc]			–		–				–					
Init_Conversation[3]	↓	/	/	/	/	/	/	/	/	/	/	/	/	/
[ok]	2													
[pc]	–													
Init_For_Incoming	↓	/	/	/	/	/	/	/	/	/	/	/	/	/
[ok]	14													
Prepare_To_Receive(C)	[pc]	[sc]	↓′	[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^									
[oi,pc]			–		–									
[sc]			–		/									
Prepare_To_Receive(F)	[pc]	[sc]	↓′	[sc]	↓′	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]			4		4									
[oi,pc]			–		–									
[sc]			–		/									
Prepare_To_Receive(S)	[pc]	[sc]	↓′	[sc]	↓′	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]			9		9									
[oi,pc]			–		–									
[sc]			–		/									
Receive(I)	[pc]	[sc]	[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]	[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]			/	–	/									
[oi,pc]			–	–	–									
[sc]			–	/	/									
Request_To_Send	[pc]	[sc]	↓′	↓′	↓′	↓′	↓′	↓′	[sc]	[sc]	↓′	↓′	↓′	[sc]
[oi,ok,pc]			–	–	–	–	–	–			–	–	–	

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

Inputs	States 1–8 and 14 are used by all conversations								Used only by conversations with *sync_level* set to CM_SYNC_POINT					
	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	Initialize-Incoming 14
Send_Data	[pc]	[sc]	↓′	[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									
[oi,pc]			–		–									
[sc]			–		/									
Send_Error	[pc]	[sc]	↓′	↓′	↓′	↓′	↓′	↓′	[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	/	/	/	/	/			/	/	/	
[oi,pc]			–	–	–	–	–	–			–	–	–	
[rb]			1^	1^	1^	1^	1^	1^			1^	1^	1^	
[rf]			1	1	1	1	1	1			1	1	1	
Set_Conv_Sec_PW	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–												
Set_Conv_Sec_Type	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–												
Set_Conv_Sec_User_ID	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–												
Set_Conv_Type	[pc]	↓	[sc]	[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]	[sc]
[ok,pc]		–												
Set_Partner_LU_Name	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–												
Set_Prep_To_Rcv_Type	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–
Set_Processing_Mode	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–
Set_Receive_Type	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–

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

Inputs	States 1–8 and 14 are used by all conversations								Used only by conversations with *sync_level* set to CM_SYNC_POINT					
	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	Initialize-Incoming 14
Set_Return_Control	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–												
Set_Send_Type	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–
Set_Sync_Level	[pc]	↓	[sc]	[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]	[sc]
[ok,pc]		–												
Test_Req_To_Send_Rcd	[pc]	[sc]	↓′	↓′	↓′	[sc]	[sc]	[sc]	↓′	↓′	[sc]	[sc]	[sc]	[sc]
[ok,pc]			–	–	–				–	–				
Wait_For_Conversation	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok] [sc,se]	/ /	% –	% –	% –	% –	% –	% –	% –	% –	% –	% –	% –	% –	% –

Effects of Calls to the SAA Resource Recovery Interface

Table 9 on page 223 shows the state transitions resulting from calls to the SAA 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_PROGRAM_STATE_CHECK

[3] 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 15.

Table 9. States and Transitions for Protected Conversations

Inputs	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	Initialize_Incoming 14
Commit call	[sc][4]	↓[6]	↓	[sc]	↓	[sc]	[sc]	[sc]	↓	↓	↓	↓	↓	↓[6]
[ok,cop,com]		–	–		3				4	1	4	3	1	–
[bo,bop,bom]		–	#		#				#	#	#	#	#	–
[sc]		–	–		–				–	–	–	–	–	–
Backout call	↓[5]	↓[6]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓[6]
[ok,bop,bom]	–	–	#	#	#	#	#	#	#	#	#	#	#	–

[4] 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.

[5] 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.

[6] Conversations in **Initialize** or **Initialize-Incoming** state are not affected by Commit and Backout calls.

State Table

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 228 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 226).
- 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 226).
- 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
- USER_CONTROL_DATA.

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."

Note: A publication that may be of interest to APPC programmers is *The APPC Resource Book* (G325-0055). It lists over 200 courses and books offered by IBM

© Copyright IBM Corp. 1988, 1993

and other companies on CPI Communications, APPC, and related topics. This book also includes extensive directories of more than 350 APPC platforms, gateways, applications, and development tools.

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 52.

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 User_Control_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: Although a CPI Communications program cannot specify LOCKS=LONG on the Prepare_To_Receive call, it can communicate with partner APPC programs that use 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 228 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 | LU 6.2 Verbs | | |

Table 10 (Page 2 of 3). Relationship of LU 6.2 Verbs to CPI Communications Calls

| CPI Communications Calls | LU 6.2 Verbs | | |

Table 10 (Page 3 of 3). Relationship of LU 6.2 Verbs to CPI Communications Calls

CPI Communications Calls	LU 6.2 Verbs																																
	GET_ATTRIBUTES	PARTNER_LU_NAME	PART_FULL_QUAL_LU_NAME	MODE_NAME	SYNC_LEVEL	CONVERSATION_STATE	CONV_CORRELATOR	**POST_ON_RECEIPT**	FILL	**PREPARE_TO_RECEIVE**	TYPE	LOCKS	**RECEIVE_AND_WAIT**	FILL	REQ_TO_SEND_RECEIVED	WHAT_RECEIVED	**RECEIVE_IMMEDIATE**	FILL	REQ_TO_SEND_RECEIVED	WHAT_RECEIVED	**REQUEST_TO_SEND**	**SEND_DATA**	REQ_TO_SEND_RECEIVED	**SEND_ERROR**	TYPE	LOG_DATA	REQ_TO_SEND_RECEIVED	**TEST**	TEST=POSTED	TEST=REQ_TO_SEND_RCVD
Starter Set																																	
Accept_Conversation																																	
Allocate																																	
Deallocate																																	
Init_Conversation											D			D				D								D							
Receive													X		X	X	X		X	X													
Send_Data																						X	X										
Advanced Function																																	
Accept_Incoming																																	
Cancel_Conversation																																	
Confirm																																	
Confirmed																																	
Convert_Incoming																																	
Convert_Outgoing																																	
Extract_Conversation_Context																																	
Extract_Conversation_State	X					X																											
Extract_Conversation_Type																																	
Extract_Max_Buffer_Size																																	
Extract_Mode_Name	X			X																													
Extract_Part_LU_Name	X	X	X																														
Extract_Security_User_ID																																	
Extract_Sync_Level	X				X																												
Extract_TP_Name																																	
Flush																																	
Initialize_For_Incoming											D			D				D								D							
Prep_To_Receive										X																							
Release_Local_TP_Name																																	
Req_To_Send																					X												
Send_Error																								X	D		X						
Set_Conv_Security_Password																																	
Set_Conv_Security_Type																																	
Set_Conv_Security_User_ID																																	
Set_Conversation_Type																																	
Set_Deallocate_Type																																	
Set_Error_Direction																								X									
Set_Fill													X	S			X	S															
Set_Log_Data																								X		S							
Set_Mode_Name																																	
Set_Part_LU_Name																																	
Set_Prep_To_Rec_Type										X	S																						
Set_Processing_Mode																																	
Set_Receive_Type													X				X																
Set_Return_Control																																	
Set_Send_Type																																	
Set_Sync_Level																																	
Set_TP_Name																																	
Specify_Local_TP_Name																																	
Test_Req_To_Send_Rcvd																													X		X		
Wait_For_Conversation																																	

Appendix E. Pseudonym Files

This appendix contains sample pseudonym files for several programming languages. These pseudonym files are provided as a usability aid for the CPI Communications programmer. Because the file names and file contents differ from system to system, the samples shown in this appendix are representative but not complete. Customized pseudonym files may be provided on many systems that use CPI Communications.

© Copyright IBM Corp. 1988, 1993

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              03/19/91 09:37:09
:group     name    = C$$CCPI
           date    = '03/18/91'  time    = '13:47:19'  refine  = N
           desc    = 'Setup CPI Communications values'.
:stmts.
;
; /* ** *** CPI Communications variable initialization ***********
;
; /* ** *** Edition SC31-6180-00 *********************************
;
; /* ** *** conversation_state ***********************************
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 14 TO CM_INITIALIZE_INCOMING_STATE;
;
; /* ** *** conversation_type ************************************
;
MOVE 0 TO CM_BASIC_CONVERSATION;
MOVE 1 TO CM_MAPPED_CONVERSATION;
;
; /* ** *** conversation_security_type ***************************
;
MOVE 0 TO CM_SECURITY_NONE;
MOVE 1 TO CM_SECURITY_SAME;
MOVE 2 TO CM_SECURITY_PROGRAM;
;
; /* ** *** return_code, conversation_return_code ****************
;
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 35 TO CM_OPERATION_INCOMPLETE;
MOVE 36 TO CM_SYSTEM_EVENT;
MOVE 37 TO CM_OPERATION_NOT_ACCEPTED;
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;
;
; /* *** data_received ******************************************
;
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;
;
; /* *** deallocate_type ****************************************
;
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;
;
; /* *** error_direction ****************************************
;
MOVE 0 TO CM_RECEIVE_ERROR;
MOVE 1 TO CM_SEND_ERROR;
;
; /* *** fill ***************************************************
;
MOVE 0 TO CM_FILL_LL;
MOVE 1 TO CM_FILL_BUFFER;
;
; /* *** prepare_to_receive_type ********************************
;
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;
;
; /* *** processing_mode ****************************************
;
MOVE 0 TO CM_BLOCKING;
MOVE 1 TO CM_NON_BLOCKING;
;
; /* *** receive_type *******************************************
;
MOVE 0 TO CM_RECEIVE_AND_WAIT;
MOVE 1 TO CM_RECEIVE_IMMEDIATE;
;
```

Application Generator Pseudonym File

```
; /* *** request_to_send_received ********************************
;
MOVE 0 TO CM_REQ_TO_SEND_NOT_RECEIVED;
MOVE 1 TO CM_REQ_TO_SEND_RECEIVED;
;
; /* *** return_code *********************************************
;
MOVE 0 TO CM_WHEN_SESSION_ALLOCATED;
MOVE 1 TO CM_IMMEDIATE;
;
; /* *** send_type ***********************************************
;
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;
;
; /* *** status_received *****************************************
;
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;
;
; /* *** status_received *****************************************
;
MOVE 0 TO CM_NONE;
MOVE 1 TO CM_CONFIRM;
MOVE 2 TO CM_SYNC_POINT;
;
;
:estmts.
:egroup.
:EZEE 330              03/19/91 09:37:16
:record    name     = C$$CPIC
           date     = '03/19/91'   time = '09:36:34'
           org      = WORKSTOR
           scope    = GLOBAL
:prol.
CSP Definitions for CPI Communications variables and values.
```

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 Communications call statements must be defined in CSP as LEVEL 77 items, and are included for the user's convenience. since CSP 3.3.0 supports the long name formats shown in the CPI Communications Specification, the definitions are shipped in this release using the same names as defined within the CPI Communications Specification. 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_INITIALIZE_INCOMING_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_SECURITY_NONE
           level    = 10        occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_ALLOCATE_FAILURE_NO_RETRY
           level    = 10        occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_SECURITY_SAME
           level    = 10        occurs   = 00001
           scope    = GLOBAL
```

Application Generator Pseudonym File

```
                    :recditem  name     = CM_ALLOCATE_FAILURE_RETRY
                               level    = 10        occurs   = 00001
                               scope    = GLOBAL
|                   :recditem  name     = CM_SECURITY_PROGRAM
|                              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_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_OPERATION_INCOMPLETE
           level    = 10        occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_SYSTEM_EVENT
           level    = 10        occurs   = 00001
           scope    = GLOBAL
:recditem  name     = CM_OPERATION_NOT_ACCEPTED
           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_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
```

Application Generator Pseudonym File

```
                    :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_BLOCKING
|                              level    = 10        occurs   = 00001
|                              scope    = GLOBAL
|                   :recditem  name     = CM_NON_BLOCKING
|                              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_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     = 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     = BUFFER
|                              level    = 77        occurs   = 00001
|                              scope    = GLOBAL
|                   :recditem  name     = BUFFER_LENGTH
|                              level    = 77        occurs   = 00001
|                              scope    = GLOBAL
|                   :recditem  name     = CONTEXT_ID
|                              level    = 77        occurs   = 00001
|                              scope    = GLOBAL
|                   :recditem  name     = CONTEXT_ID_LENGTH
|                              level    = 77        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     = SECURITY_PASSWORD
|                              level    = 77        occurs   = 00001
|                              scope    = GLOBAL
```

Application Generator Pseudonym File

```
|                     :recditem  name     = SECURITY_PASSWORD_LENGTH
|                                level    = 77        occurs   = 00001
|                                scope    = GLOBAL
|                     :recditem  name     = CONVERSATION_SECURITY_TYPE
|                                level    = 77        occurs   = 00001
|                                scope    = GLOBAL
|                     :recditem  name     = SECURITY_USER_ID
|                                level    = 77        occurs   = 00001
|                                scope    = GLOBAL
|                     :recditem  name     = SECURITY_USER_ID_LENGTH
|                                level    = 77        occurs   = 00001
|                                scope    = GLOBAL
|                     :recditem  name     = CONVERSATION_RETURN_CODE
|                                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     = MAXIMUM_BUFFER_SIZE
|                                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     = PROCESSING_MODE
|                                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.
:item      name  = CM_INITIALIZE_STATE
           date  = '03/18/91'  time     = '14:22:42' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_SEND_STATE
           date  = '03/18/91'  time     = '14:22:42' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_RECEIVE_STATE
           date  = '03/18/91'  time     = '14:22:43' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_SEND_PENDING_STATE
           date  = '03/18/91'  time     = '14:22:43' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_CONFIRM_STATE
           date  = '03/18/91'  time     = '14:22:43' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_CONFIRM_SEND_STATE
           date  = '03/18/91'  time     = '14:22:43' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_CONFIRM_DEALLOCATE_STATE
           date  = '03/18/91'  time     = '14:22:43' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_DEFER_RECEIVE_STATE
           date  = '03/18/91'  time     = '14:22:43' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
```

Application Generator Pseudonym File

```
                    :item      name  = CM_DEFER_DEALLOCATE_STATE
                               date  = '03/18/91'  time     = '14:22:43' type     = BIN
                               bytes = 00004       decimals = 00         evensql  = N
                    :eitem.
                    :item      name  = CM_SYNC_POINT_STATE
                               date  = '03/18/91'  time     = '14:22:43' type     = BIN
                               bytes = 00004       decimals = 00         evensql  = N
                    :eitem.
                    :item      name  = CM_SYNC_POINT_SEND_STATE
                               date  = '03/18/91'  time     = '14:22:43' type     = BIN
                               bytes = 00004       decimals = 00         evensql  = N
                    :eitem.
                    :item      name  = CM_SYNC_POINT_DEALLOCATE_STATE
                               date  = '03/18/91'  time     = '14:22:43' type     = BIN
                               bytes = 00004       decimals = 00         evensql  = N
                    :eitem.
|                   :item      name  = CM_INITIALIZE_INCOMING_STATE
|                              date  = '10/31/92'  time     = '24:00:00' type     = BIN
|                              bytes = 00004       decimals = 00         evensql  = N
|                   :eitem.
                    :item      name  = CM_BASIC_CONVERSATION
                               date  = '02/07/90'  time     = '10:54:59' type     = BIN
                               bytes = 00004       decimals = 00         evensql  = N
                    :eitem.
                    :item      name  = CM_MAPPED_CONVERSATION
                               date  = '02/07/90'  time     = '11:27:11' type     = BIN
                               bytes = 00004       decimals = 00         evensql  = N
                    :eitem.
|                   :item      name  = CM_SECURITY_NONE
|                              date  = '10/31/92'  time     = '24:00:00' type     = BIN
|                              bytes = 00004       decimals = 00         evensql  = N
|                   :eitem.
|                   :item      name  = CM_SECURITY_SAME
|                              date  = '10/31/92'  time     = '24:00:00' type     = BIN
|                              bytes = 00004       decimals = 00         evensql  = N
|                   :eitem.
|                   :item      name  = CM_SECURITY_PROGRAM
|                              date  = '10/31/92'  time     = '24:00:00' type     = BIN
|                              bytes = 00004       decimals = 00         evensql  = N
|                   :eitem.
                    :item      name  = CM_OK
                               date  = '02/07/90'  time     = '10:54:59' type     = BIN
                               bytes = 00004       decimals = 00         evensql  = N
                    :eitem.
                    :item      name  = CM_ALLOCATE_FAILURE_NO_RETRY
                               date  = '02/07/90'  time     = '10:55:00' type     = BIN
                               bytes = 00004       decimals = 00         evensql  = N
                    :eitem.
                    :item      name  = CM_ALLOCATE_FAILURE_RETRY
                               date  = '02/07/90'  time     = '11:28:53' type     = BIN
                               bytes = 00004       decimals = 00         evensql  = N
                    :eitem.
                    :item      name  = CM_CONVERSATION_TYPE_MISMATCH
                               date  = '03/19/91'  time     = '09:24:13' type     = BIN
                               bytes = 00004       decimals = 00         evensql  = N
                    :eitem.
                    :item      name  = CM_PIP_NOT_SPECIFIED_CORRECTLY
                               date  = '03/19/91'  time     = '09:24:13' type     = BIN
                               bytes = 00004       decimals = 00         evensql  = N
                    :eitem.
                    :item      name  = CM_SECURITY_NOT_VALID
                               date  = '02/07/90'  time     = '10:55:00' type     = BIN
                               bytes = 00004       decimals = 00         evensql  = N
                    :eitem.
```

```
:item      name  = CM_SYNC_LVL_NOT_SUPPORTED_LU
           date  = '03/18/91'  time     = '14:22:43' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_SYNC_LVL_NOT_SUPPORTED_PGM
           date  = '02/07/90'  time     = '10:55:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_TPN_NOT_RECOGNIZED
           date  = '02/07/90'  time     = '10:55:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_TP_NOT_AVAILABLE_NO_RETRY
           date  = '02/07/90'  time     = '10:55:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_TP_NOT_AVAILABLE_RETRY
           date  = '02/07/90'  time     = '10:55:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_DEALLOCATED_ABEND
           date  = '02/07/90'  time     = '10:55:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_DEALLOCATED_NORMAL
           date  = '02/07/90'  time     = '10:55:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_PARAMETER_ERROR
           date  = '02/07/90'  time     = '10:55:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_PRODUCT_SPECIFIC_ERROR
           date  = '02/08/90'  time     = '06:31:36' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_PROGRAM_ERROR_NO_TRUNC
           date  = '02/07/90'  time     = '10:55:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_PROGRAM_ERROR_PURGING
           date  = '02/07/90'  time     = '10:55:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_PROGRAM_ERROR_TRUNC
           date  = '02/07/90'  time     = '10:55:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_PROGRAM_PARAMETER_CHECK
           date  = '02/07/90'  time     = '10:55:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_PROGRAM_STATE_CHECK
           date  = '02/07/90'  time     = '10:55:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_RESOURCE_FAILURE_NO_RETRY
           date  = '02/07/90'  time     = '10:55:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_RESOURCE_FAILURE_RETRY
           date  = '02/07/90'  time     = '10:55:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
```

```
:item      name  = CM_UNSUCCESSFUL
           date  = '02/07/90'  time     = '10:55:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_DEALLOCATED_ABEND_SVC
           date  = '03/18/91'  time     = '14:22:43' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_DEALLOCATED_ABEND_TIMER
           date  = '03/18/91'  time     = '14:22:43' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_SVC_ERROR_NO_TRUNC
           date  = '03/18/91'  time     = '14:22:43' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_SVC_ERROR_PURGING
           date  = '03/18/91'  time     = '14:22:43' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_SVC_ERROR_TRUNC
           date  = '03/18/91'  time     = '14:22:43' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_OPERATION_INCOMPLETE
           date  = '10/31/92'  time     = '24:00:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_SYSTEM_EVENT
           date  = '10/31/92'  time     = '24:00:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_OPERATION_NOT_ACCEPTED
           date  = '10/31/92'  time     = '24:00:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_TAKE_BACKOUT
           date  = '03/18/91'  time     = '14:22:43' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_DEALLOCATED_ABEND_BO
           date  = '03/18/91'  time     = '14:22:43' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_DEALLOCATED_ABEND_SVC_BO
           date  = '03/18/91'  time     = '14:22:43' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_DEALLOCATED_ABEND_TIMER_BO
           date  = '03/18/91'  time     = '14:22:43' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_RESOURCE_FAIL_NO_RETRY_BO
           date  = '03/18/91'  time     = '14:22:43' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_RESOURCE_FAILURE_RETRY_BO
           date  = '03/18/91'  time     = '14:22:43' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_DEALLOCATED_NORMAL_BO
           date  = '03/18/91'  time     = '14:22:43' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
```

```
:item      name  = CM_NO_DATA_RECEIVED
           date  = '02/07/90'  time     = '10:55:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_DATA_RECEIVED
           date  = '02/07/90'  time     = '10:55:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_COMPLETE_DATA_RECEIVED
           date  = '02/07/90'  time     = '10:55:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_INCOMPLETE_DATA_RECEIVED
           date  = '02/07/90'  time     = '10:55:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_DEALLOCATE_SYNC_LEVEL
           date  = '02/07/90'  time     = '10:55:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_DEALLOCATE_FLUSH
           date  = '02/07/90'  time     = '10:55:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_DEALLOCATE_CONFIRM
           date  = '02/07/90'  time     = '10:55:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_DEALLOCATE_ABEND
           date  = '02/07/90'  time     = '10:55:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_RECEIVE_ERROR
           date  = '02/07/90'  time     = '10:55:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_SEND_ERROR
           date  = '02/07/90'  time     = '10:55:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_FILL_LL
           date  = '02/07/90'  time     = '10:55:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_FILL_BUFFER
           date  = '02/07/90'  time     = '10:55:01' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_PREP_TO_RECEIVE_SYNC_LEVEL
           date  = '02/07/90'  time     = '10:55:01' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_PREP_TO_RECEIVE_FLUSH
           date  = '02/07/90'  time     = '10:55:01' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_PREP_TO_RECEIVE_CONFIRM
           date  = '02/07/90'  time     = '10:55:01' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_BLOCKING
           date  = '10/31/92'  time     = '24:00:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
```

Application Generator Pseudonym File

```
:item      name  = CM_NON_BLOCKING
           date  = '10/31/92'  time     = '24:00:00' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_RECEIVE_AND_WAIT
           date  = '02/07/90'  time     = '10:55:01' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_RECEIVE_IMMEDIATE
           date  = '02/07/90'  time     = '10:55:01' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_REQ_TO_SEND_NOT_RECEIVED
           date  = '02/07/90'  time     = '10:55:01' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_REQ_TO_SEND_RECEIVED
           date  = '02/07/90'  time     = '10:55:01' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_WHEN_SESSION_ALLOCATED
           date  = '02/07/90'  time     = '10:55:01' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_IMMEDIATE
           date  = '02/07/90'  time     = '10:55:01' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_BUFFER_DATA
           date  = '02/07/90'  time     = '10:55:01' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_SEND_AND_FLUSH
           date  = '02/07/90'  time     = '10:55:01' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_SEND_AND_CONFIRM
           date  = '02/07/90'  time     = '10:55:01' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_SEND_AND_PREP_TO_RECEIVE
           date  = '02/07/90'  time     = '10:55:02' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_SEND_AND_DEALLOCATE
           date  = '02/07/90'  time     = '10:55:02' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_NO_STATUS_RECEIVED
           date  = '02/07/90'  time     = '10:55:02' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_SEND_RECEIVED
           date  = '02/07/90'  time     = '10:55:02' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_CONFIRM_RECEIVED
           date  = '02/07/90'  time     = '10:55:02' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CM_CONFIRM_SEND_RECEIVED
           date  = '02/07/90'  time     = '10:55:02' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
```

```
                    :item      name  = CM_CONFIRM_DEALLOC_RECEIVED
                               date  = '02/07/90'  time     = '10:55:02' type     = BIN
                               bytes = 00004       decimals = 00         evensql  = N
                    :eitem.
                    :item      name  = CM_TAKE_COMMIT
                               date  = '03/18/91'  time     = '14:22:43' type     = BIN
                               bytes = 00004       decimals = 00         evensql  = N
                    :eitem.
                    :item      name  = CM_TAKE_COMMIT_SEND
                               date  = '03/18/91'  time     = '14:22:43' type     = BIN
                               bytes = 00004       decimals = 00         evensql  = N
                    :eitem.
                    :item      name  = CM_TAKE_COMMIT_DEALLOCATE
                               date  = '03/18/91'  time     = '14:22:43' type     = BIN
                               bytes = 00004       decimals = 00         evensql  = N
                    :eitem.
                    :item      name  = CM_NONE
                               date  = '02/07/90'  time     = '10:55:02' type     = BIN
                               bytes = 00004       decimals = 00         evensql  = N
                    :eitem.
                    :item      name  = CM_CONFIRM
                               date  = '02/07/90'  time     = '10:55:02' type     = BIN
                               bytes = 00004       decimals = 00         evensql  = N
                    :eitem.
                    :item      name  = CM_SYNC_POINT
                               date  = '03/18/91'  time     = '14:22:43' type     = BIN
                               bytes = 00004       decimals = 00         evensql  = N
                    :eitem.
|                   :item      name  = BUFFER
|                              date  = '02/07/90'  time     = '10:55:02' type     = CHA
|                              bytes = 00000       decimals = 00         evensql  = N
|                   :eitem.
|                   :item      name  = BUFFER_LENGTH
|                              date  = '02/07/90'  time     = '10:55:02' type     = BIN
|                              bytes = 00004       decimals = 00         evensql  = N
|                   :eitem.
|                   :item      name  = CONTEXT_ID
|                              date  = '02/07/90'  time     = '10:55:02' type     = CHA
|                              bytes = 00008       decimals = 00         evensql  = N
|                   :eitem.
|                   :item      name  = CONTEXT_ID_LENGTH
|                              date  = '02/01/93'  time     = '10:55:02' type     = CHA
|                              bytes = 00004       decimals = 00         evensql  = N
|                   :eitem.
                    :item      name  = CONVERSATION_ID
                               date  = '02/07/90'  time     = '10:55:02' type     = CHA
                               bytes = 00008       decimals = 00         evensql  = N
                    :eitem.
                    :item      name  = CONVERSATION_STATE
                               date  = '03/18/91'  time     = '14:22:44' type     = BIN
                               bytes = 00004       decimals = 00         evensql  = N
                    :eitem.
                    :item      name  = CONVERSATION_TYPE
                               date  = '02/08/90'  time     = '05:43:53' type     = BIN
                               bytes = 00004       decimals = 00         evensql  = N
                    :eitem.
|                   :item      name  = SECURITY_PASSWORD
|                              date  = '02/01/93'  time     = '05:43:53' type     = BIN
|                              bytes = 00010       decimals = 00         evensql  = N
|                   :eitem.
|                   :item      name  = SECURITY_PASSWORD_LENGTH
|                              date  = '02/01/93'  time     = '05:43:53' type     = BIN
|                              bytes = 00004       decimals = 00         evensql  = N
|                   :eitem.
```

Application Generator Pseudonym File

```
:item      name  = CONVERSATION_SECURITY_TYPE
           date  = '02/08/90'  time     = '05:43:53' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = SECURITY_USER_ID
           date  = '02/01/93'  time     = '05:43:53' type     = BIN
           bytes = 00010       decimals = 00         evensql  = N
:eitem.
:item      name  = SECURITY_USER_ID_LENGTH
           date  = '02/01/93'  time     = '05:43:53' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = CONVERSATION_RETURN_CODE
           date  = '02/01/93'  time     = '05:43:53' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = RETURN_CODE
           date  = '02/07/90'  time     = '09:39:48' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = DATA_RECEIVED
           date  = '02/07/90'  time     = '09:39:48' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = DEALLOCATE_TYPE
           date  = '02/07/90'  time     = '10:55:02' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = ERROR_DIRECTION
           date  = '02/07/90'  time     = '09:39:49' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = FILL
           date  = '02/07/90'  time     = '09:39:49' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = LOG_DATA
           date  = '02/07/90'  time     = '09:39:49' type     = CHA
           bytes = 00512       decimals = 00         evensql  = N
:eitem.
:item      name  = LOG_DATA_LENGTH
           date  = '02/07/90'  time     = '10:55:02' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = MAXIMUM_BUFFER_SIZE
           date  = '02/07/90'  time     = '10:55:02' type     = CHA
           bytes = 00008       decimals = 00         evensql  = N
:eitem.
:item      name  = MODE_NAME
           date  = '02/07/90'  time     = '10:55:02' type     = CHA
           bytes = 00008       decimals = 00         evensql  = N
:eitem.
:item      name  = MODE_NAME_LENGTH
           date  = '02/07/90'  time     = '10:55:02' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = PARTNER_LU_NAME
           date  = '02/07/90'  time     = '10:55:02' type     = CHA
           bytes = 00017       decimals = 00         evensql  = N
:eitem.
:item      name  = PARTNER_LU_NAME_LENGTH
           date  = '02/07/90'  time     = '10:55:02' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
```

```
:item      name  = PREPARE_TO_RECEIVE_TYPE
           date  = '02/07/90'  time     = '10:55:02' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = PROCESSING_MODE
           date  = '02/01/93'  time     = '10:55:02' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = RECEIVED_LENGTH
           date  = '02/07/90'  time     = '10:55:02' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = RECEIVE_TYPE
           date  = '02/07/90'  time     = '10:55:02' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = REQUESTED_LENGTH
           date  = '02/07/90'  time     = '10:55:02' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = REQUEST_TO_SEND_RECEIVED
           date  = '02/09/90'  time     = '05:17:34' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = RETURN_CONTROL
           date  = '02/07/90'  time     = '10:55:03' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = SEND_LENGTH
           date  = '02/07/90'  time     = '10:55:03' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = SEND_TYPE
           date  = '02/07/90'  time     = '10:55:03' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = STATUS_RECEIVED
           date  = '02/07/90'  time     = '10:55:03' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = SYNC_LEVEL
           date  = '02/07/90'  time     = '10:55:03' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
:item      name  = SYM_DEST_NAME
           date  = '02/07/90'  time     = '10:55:03' type     = CHA
           bytes = 00008       decimals = 00         evensql  = N
:eitem.
:item      name  = TP_NAME
           date  = '02/07/90'  time     = '10:55:03' type     = CHA
           bytes = 00064       decimals = 00         evensql  = N
:eitem.
:item      name  = TP_NAME_LENGTH
           date  = '02/07/90'  time     = '10:55:03' type     = BIN
           bytes = 00004       decimals = 00         evensql  = N
:eitem.
```

C Pseudonym File (CMC or CPIC.H)

CPIC 1.0 and CPIC 1.1 contained a pseudonym file for C named CMC. X/Open introduced a pseudonym file named CPIC.H. The updated pseudonym file merges CMC and CPIC.H for CPIC 1.2.

```
/*
 *
 * Edition SC31-6180-00

 * NOTE:
 * Before you use this file, you must set a supported operating
 * system constant.  Search on SYSTEM to find the list of supported
 * constants and how to set the appropriate one.
 */


/*
 * This file is organized as follows:
 *  - product-specific preprocessor directives that must be before all
 *    other declarations
 *  - Base CPI-C constants and type definitions
 *  - Base CPI-C function prototypes
 *  - Product-specific constants and type definitions (enclosed in #if/#endif)
 *  - Product-specific function prototypes (enclosed in #if/#endif)
 */

#ifndef _cpic_h
#define _cpic_h

/*
 * SYSTEM should be changed in the following #define
 * to an appropriate value for your platform.
 * The following values are handled by this file:
 *    CM_AIX
 *    CM_DOS
 *    CM_MVS
 *    CM_OS2
 *    CM_OS400
 *    CM_VM
 *
 * This is necessary for the proper setting of
 * CM_ENTRY and CM_PTR below.
 */


#if ! ( defined(CM_AIX) || defined(CM_DOS) || defined(CM_MVS) || \
        defined(CM_OS2) || defined(CM_OS400) || defined(CM_VM)   \
      )
/*
 * Define the system ONLY if no system has been defined by the program.
 * This avoids duplicate macro definition warnings and allows this
 * include file to be moved to other platforms more easily.
 *
 * If you choose to move this include file to another platform,
 * we recommend that you define the CM_ constant for that platform
 * externally on the compiler command line.
 */

#define SYSTEM

#endif
```

```
                        /*
                         * 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




                        #ifdef   CM_OS2
                        #ifdef   __32BIT__
                        #ifndef  ES32TO16
                        #define  ES32TO16
                        #endif
                        #endif
                        #endif



                        #if   defined( CM_DOS ) || defined( CM_OS2 )

                        #     define CM_ENTRY extern void pascal far _loadds
                        #     define CM_PTR far *

                        #elif defined( CM_MVS ) || defined( CM_OS400 ) || \
                              defined( CM_VM ) || defined(AIX)

                        #     define CM_ENTRY extern void
                        #     define CM_PTR *

                        #else

                        /*
                         * If we have gotten this far, a supported system constant has not
                         * been defined.  We will produce a syntax error on the next line
                         * that should help the user identify the problem.
                         *
                         * To correct this problem, a system constant should be defined
                         * either above in this file, or externally with a compiler option.
                         */
                        INTENTIONAL_SYNTAX_ERROR define_a_system_in_cpic_h();

                        #endif



                        #ifdef ES32TO16

                        /*
                         * The IBM C Set/2 32 bit compiler for OS/2 2.0 requires different
                         * definitions for entry points, pointers, and structures.
                         * These definitions help the compiler perform the translations from
                         * 32-bit addresses to segmented addresses.
                         */

                        #   ifdef CM_PTR
                        #      undef CM_PTR
```

```
#   endif
#   ifdef CM_ENTRY
#      undef CM_ENTRY
#   endif
#   ifdef CM_INT32
#      undef CM_INT32
#   endif
# define CM_PTR * _Seg16
# define CM_ENTRY extern void
# define CM_INT32 signed int

# define STRUCT16 _Packed struct
# pragma seg16(SIDE_INFO)
#else
# define STRUCT16 struct
#endif

#if defined(CM_OS400) || defined(CM_MVS)
#define cmacci CMACCI
#define cmaccp CMACCP
#define cmallc CMALLC
#define cmcanc CMCANC
#define cmcfm  CMCFM
#define cmcfmd CMCFMD
#define cmcnvi CMCNVI
#define cmcnvo CMCNVO
#define cmdeal CMDEAL
#define cmecs  CMECS
#define cmect  CMECT
#define cmectx CMECTX
#define cmembs CMEMBS
#define cmemn  CMEMN
#define cmepln CMEPLN
#define cmesl  CMESL
#define cmesui CMESUI
#define cmetpn CMETPN
#define cmflus CMFLUS
#define cminic CMINIC
#define cminit CMINIT
#define cminit CMINIT
#define cmptr  CMPTR
#define cmrcv  CMRCV
#define cmrltp CMRLTP
#define cmrts  CMRTS
#define cmscsp CMSCSP
#define cmscst CMSCST
#define cmscsu CMSCSU
#define cmsct  CMSCT
#define cmsdt  CMSDT
#define cmsed  CMSED
#define cmsend CMSEND
#define cmserr CMSERR
#define cmsf   CMSF
#define cmsld  CMSLD
#define cmsltp CMSLTP
#define cmsmn  CMSMN
#define cmspln CMSPLN
#define cmspm  CMSPM
#define cmsptr CMSPTR
#define cmsrc  CMSRC
#define cmsrt  CMSRT
#define cmssl  CMSSL
#define cmsst  CMSST
#define cmstpn CMSTPN
```

```
#define cmtrts CMTRTS
#define cmwait CMWAIT
#endif



/*
 *  - Base CPI-C constants and type definitions
 */

/* conversation ID */
typedef char CONVERSATION_ID [8];
typedef char CONTEXT_ID [32];
typedef char SECURITY_PASSWORD [10];
typedef char SECURITY_USER_ID [10];

typedef CM_INT32 CM_BUFFER_LENGTH;
typedef CM_INT32 CM_CONTEXT_ID_LENGTH;
typedef CM_INT32 CM_CONVERSATION_RETURN_CODE;
typedef CM_INT32 CM_SECURITY_PASSWORD_LENGTH;
typedef CM_INT32 CM_CONVERSATION_SECURITY_TYPE;
typedef CM_INT32 CM_SECURITY_USER_ID_LENGTH;
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_MAXIMUM_BUFFER_SIZE;
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;
typedef CM_INT32 CM_PROCESSING_MODE;

/* X/open typedefs for compatibility  */
typedef CM_INT32 CONVERSATION_STATE;
typedef CM_INT32 CONVERSATION_TYPE;
typedef CM_INT32 CONVERSATION_SECURITY_TYPE;
typedef CM_INT32 DATA_RECEIVED_TYPE;
typedef CM_INT32 DEALLOCATE_TYPE;
typedef CM_INT32 ERROR_DIRECTION;
typedef CM_INT32 FILL;
typedef CM_INT32 PREPARE_TO_RECEIVE_TYPE;
typedef CM_INT32 RECEIVE_TYPE;
typedef CM_INT32 REQUEST_TO_SEND_RECEIVED;
typedef CM_INT32 CM_RETCODE;
typedef CM_INT32 RETURN_CONTROL;
typedef CM_INT32 SEND_TYPE;
typedef CM_INT32 STATUS_RECEIVED;
typedef CM_INT32 SYNC_LEVEL;
typedef CM_INT32 PROCESSING_MODE;

/*
 * Enumerated data types (enum) have not been used for the
 * constant values because the default type for an enum
 * is 'int'.  This causes type conflicts on compilers where
 * int is not the same size as CM_INT32.
 */
```

```
|                       /*  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
|                       #define CM_INITIALIZE_INCOMING_STATE    (CM_CONVERSATION_STATE) 14


|                       /*  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


|                       /*  processing_mode values  */

|                       #define CM_BLOCKING                     (CM_PROCESSING_MODE) 0
|                       #define CM_NON_BLOCKING                 (CM_PROCESSING_MODE) 1
```

```
|                   /*  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_OPERATION_INCOMPLETE         (CM_RETURN_CODE) 35   /* CPIC 1.2 */
|                   #define CM_SYSTEM_EVENT                 (CM_RETURN_CODE) 36   /* CPIC 1.2 */
|                   #define CM_OPERATION_NOT_ACCEPTED       (CM_RETURN_CODE) 37   /* CPIC 1.2 */
|                   #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
```

C Pseudonym File

```
#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

/* conversation_security_type values */

#define CM_SECURITY_NONE                (CM_CONVERSATION_SECURITY_TYPE) 0
#define CM_SECURITY_SAME                (CM_CONVERSATION_SECURITY_TYPE) 1
#define CM_SECURITY_PROGRAM             (CM_CONVERSATION_SECURITY_TYPE) 2

/* processing mode of conversation */
#define CM_BLOCKING                     (CM_PROCESSING_MODE) 0
#define CM_NON_BLOCKING                 (CM_PROCESSING_MODE) 1


/*
 *  - Base CPI-C function prototypes
 */

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

CM_ENTRY cmacci(unsigned char CM_PTR,            /* conversation_ID        */
                CM_RETURN_CODE CM_PTR);          /* return_code            */
CM_ENTRY cmaccp(unsigned char CM_PTR,            /* conversation_ID        */
                CM_RETURN_CODE CM_PTR);          /* return_code            */
CM_ENTRY cmallc(unsigned char CM_PTR,            /* conversation_ID        */
                CM_RETURN_CODE CM_PTR);          /* return_code            */
CM_ENTRY cmcanc(unsigned char CM_PTR,            /* conversation_ID        */
                CM_RETURN_CODE CM_PTR);          /* return_code            */
CM_ENTRY cmcfm( unsigned char CM_PTR,            /* conversation_ID        */
                CM_REQUEST_TO_SEND_RECEIVED CM_PTR,/*request_to_send_received*/
                CM_RETURN_CODE CM_PTR);          /* return_code            */
CM_ENTRY cmcfmd(unsigned char CM_PTR,            /* conversation_ID        */
                CM_RETURN_CODE CM_PTR);          /* return_code            */
CM_ENTRY cmcnvi(unsigned char CM_PTR,            /* buffer                 */
                CM_INT32 CM_PTR,                 /* buffer_length          */
                CM_RETURN_CODE CM_PTR);          /* return_code            */
CM_ENTRY cmcnvo(unsigned char CM_PTR,            /* buffer                 */
                CM_INT32 CM_PTR,                 /* buffer_length          */
                CM_RETURN_CODE CM_PTR);          /* return_code            */
CM_ENTRY cmdeal(unsigned char CM_PTR,            /* conversation_ID        */
                CM_RETURN_CODE CM_PTR);          /* return_code            */
```

```
| /* Extract_Conversation_State is not supported on AS/400 or MVS */
| #if !(defined(CM_OS400) || defined(CM_MVS))
| CM_ENTRY cmecs( unsigned char CM_PTR,              /* conversation_ID        */
|                 CM_CONVERSATION_STATE CM_PTR,     /* conversation_state     */
|                 CM_RETURN_CODE CM_PTR);           /* return_code            */
| #endif

| CM_ENTRY cmect( unsigned char CM_PTR,              /* conversation_ID        */
|                 CM_CONVERSATION_TYPE CM_PTR,      /* conversation_type      */
|                 CM_RETURN_CODE CM_PTR);           /* return_code            */
| CM_ENTRY cmectx(unsigned char CM_PTR,              /* conversation_ID        */
|                 unsigned char CM_PTR,             /* context_ID             */
|                 CM_INT32 CM_PTR,                  /* context_ID_length      */
|                 CM_RETURN_CODE CM_PTR);           /* return_code            */
| CM_ENTRY cmembs(CM_INT32 CM_PTR,                  /* maximum_buffer_size    */
|                 CM_RETURN_CODE 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_RETURN_CODE 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_RETURN_CODE CM_PTR);           /* return_code            */
| CM_ENTRY cmesl( unsigned char CM_PTR,             /* conversation_ID        */
|                 CM_SYNC_LEVEL CM_PTR,             /* sync_level             */
|                 CM_RETURN_CODE CM_PTR);           /* return_code            */
| CM_ENTRY cmesui(unsigned char  CM_PTR,            /* conversation_ID        */
|                 unsigned char  CM_PTR,            /* user_ID                */
|                 CM_INT32 CM_PTR,                  /* user_ID_length         */
|                 CM_RETURN_CODE CM_PTR);           /* return_code            */
| CM_ENTRY cmetpn(unsigned char CM_PTR,             /* conversation_ID        */
|                 unsigned char CM_PTR,             /* TP_name                */
|                 CM_INT32 CM_PTR,                  /* TP_name_length         */
|                 CM_RETURN_CODE CM_PTR);           /* return_code            */
| CM_ENTRY cmflus(unsigned char CM_PTR,             /* conversation_ID        */
|                 CM_RETURN_CODE CM_PTR);           /* return_code            */
| CM_ENTRY cminic(unsigned char CM_PTR,             /* conversation_ID        */
|                 CM_RETURN_CODE CM_PTR);           /* return_code            */
| CM_ENTRY cminit(unsigned char CM_PTR,             /* conversation_ID        */
|                 unsigned char CM_PTR,             /* sym_dest_name          */
|                 CM_RETURN_CODE CM_PTR);           /* return_code            */
| CM_ENTRY cmptr( unsigned char CM_PTR,             /* conversation_ID        */
|                 CM_RETURN_CODE 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_DATA_RECEIVED_TYPE CM_PTR,     /* data_received          */
|                 CM_INT32 CM_PTR,                  /* received_length        */
|                 CM_STATUS_RECEIVED CM_PTR,        /* status_received        */
|                 CM_REQUEST_TO_SEND_RECEIVED CM_PTR,/*request_to_send_received*/
|                 CM_RETURN_CODE CM_PTR);           /* return_code            */
| CM_ENTRY cmrltp(unsigned char CM_PTR,             /* TP_name                */
|                 CM_INT32 CM_PTR,                  /* TP_name_length         */
|                 CM_RETURN_CODE CM_PTR);           /* return_code            */
| CM_ENTRY cmrts( unsigned char CM_PTR,             /* conversation_ID        */
|                 CM_RETURN_CODE CM_PTR);           /* return_code            */
| CM_ENTRY cmscsp(unsigned char  CM_PTR,            /* conversation_ID        */
|                 unsigned char  CM_PTR,            /* password               */
|                 CM_INT32 CM_PTR,                  /* password_length        */
|                 CM_RETURN_CODE CM_PTR);           /* return_code            */
| CM_ENTRY cmscst(unsigned char  CM_PTR,            /* conversation_ID        */
|                 CM_INT32 CM_PTR,                  /* conv_security_type     */
|                 CM_RETURN_CODE CM_PTR);           /* return_code            */
```

C Pseudonym File

```
CM_ENTRY cmscsu(unsigned char  CM_PTR,             /* conversation_ID        */
                unsigned char  CM_PTR,             /* user_ID                */
                CM_INT32 CM_PTR,                   /* user_ID_length         */
                CM_RETURN_CODE CM_PTR);            /* return_code            */
CM_ENTRY cmsct( unsigned char CM_PTR,              /* conversation_ID        */
                CM_CONVERSATION_TYPE CM_PTR,       /* conversation_type      */
                CM_RETURN_CODE CM_PTR);            /* return_code            */
CM_ENTRY cmsdt( unsigned char CM_PTR,              /* conversation_ID        */
                CM_DEALLOCATE_TYPE CM_PTR,         /* deallocate_type        */
                CM_RETURN_CODE CM_PTR);            /* return_code            */
CM_ENTRY cmsed( unsigned char CM_PTR,              /* conversation_ID        */
                CM_ERROR_DIRECTION CM_PTR,         /* error_direction        */
                CM_RETURN_CODE 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_REQUEST_TO_SEND_RECEIVED CM_PTR,/*request_to_send_received*/
                CM_RETURN_CODE CM_PTR);            /* return_code            */
CM_ENTRY cmserr(unsigned char CM_PTR,              /* conversation_ID        */
                CM_REQUEST_TO_SEND_RECEIVED CM_PTR,/*request_to_send_received*/
                CM_RETURN_CODE CM_PTR);            /* return_code            */
CM_ENTRY cmsf(  unsigned char CM_PTR,              /* conversation_ID        */
                CM_FILL CM_PTR,                    /* fill                   */
                CM_RETURN_CODE 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_RETURN_CODE CM_PTR);            /* return_code            */
CM_ENTRY cmsltp(unsigned char CM_PTR,              /* TP_name                */
                CM_INT32 CM_PTR,                   /* TP_name_length         */
                CM_RETURN_CODE 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_RETURN_CODE 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_RETURN_CODE CM_PTR);            /* return_code            */
CM_ENTRY cmspm( unsigned char CM_PTR,              /* conversation_ID        */
                CM_PROCESSING_MODE CM_PTR,         /* processing_mode        */
                CM_RETURN_CODE CM_PTR);            /* return_code            */
CM_ENTRY cmsptr(unsigned char CM_PTR,              /* conversation_ID        */
                CM_PREPARE_TO_RECEIVE_TYPE CM_PTR, /* prepare_to_receive_type*/
                CM_RETURN_CODE CM_PTR);            /* return_code            */
CM_ENTRY cmsrc( unsigned char CM_PTR,              /* conversation_ID        */
                CM_RETURN_CONTROL CM_PTR,          /* return_control         */
                CM_RETURN_CODE CM_PTR);            /* return_code            */
CM_ENTRY cmsrt( unsigned char CM_PTR,              /* conversation_ID        */
                CM_RECEIVE_TYPE CM_PTR,            /* receive_type           */
                CM_RETURN_CODE CM_PTR);            /* return_code            */
CM_ENTRY cmssl( unsigned char CM_PTR,              /* conversation_ID        */
                CM_INT32 CM_PTR,                   /* sync_level             */
                CM_RETURN_CODE CM_PTR);            /* return_code            */
CM_ENTRY cmsst( unsigned char CM_PTR,              /* conversation_ID        */
                CM_SEND_TYPE CM_PTR,               /* send_type              */
                CM_RETURN_CODE 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_RETURN_CODE CM_PTR);            /* return_code            */
CM_ENTRY cmtrts(unsigned char CM_PTR,              /* conversation_ID        */
                CM_REQUEST_TO_SEND_RECEIVED CM_PTR,/*request_to_send_received*/
                CM_RETURN_CODE CM_PTR);            /* return_code            */
```

```
CM_ENTRY cmwait(unsigned char CM_PTR,             /* conversation_ID       */
                CM_RETURN_CODE CM_PTR,           /* conversation_ret_code */
                CM_RETURN_CODE CM_PTR);          /* return_code           */



/*
 * X/Open provides security calls in the base.
 * Since some SAA platforms already support the security calls through
 * product-specific extensions, we'll provide macros here to help in
 * porting X/Open applications to these platforms.
 */
#if defined(CM_OS2) || defined(CM_VM) || defined(CM_DOS) || defined(CM_AIX)
#define cmscsu(v1,v2,v3,v4) xcscsu(v1,v2,v3,v4)
#define cmscsp(v1,v2,v3,v4) xcscsp(v1,v2,v3,v4)
#define cmscst(v1,v2,v3)    xcscst(v1,v2,v3)
#if !defined(CM_DOS)
/*
 * Networking Services/DOS does not provide extract security calls.
 */
#define cmecsu(v1,v2,v3,v4) xcecsu(v1,v2,v3,v4)
#endif
#endif



#ifdef __cplusplus
}
#endif /* __cplusplus */

/*
 *  - Product-specific constants and type definitions
 */

/*
 *    #pragma linkage directives
 *
 * Note: For OS/400, routine names must be all uppercase.
 */
#if defined(CM_VM)
#     pragma linkage (cmacci, OS)
#     pragma linkage (cmaccp, OS)
#     pragma linkage (cmallc, OS)
#     pragma linkage (cmcanc, OS)
#     pragma linkage (cmcfm,  OS)
#     pragma linkage (cmcfmd, OS)
#     pragma linkage (cmcnvi, OS)
#     pragma linkage (cmcnvo, OS)
#     pragma linkage (cmdeal, OS)
#     pragma linkage (cmecs,  OS)
#     pragma linkage (cmect,  OS)
#     pragma linkage (cmectx, OS)
#     pragma linkage (cmembs, OS)
#     pragma linkage (cmemn,  OS)
#     pragma linkage (cmepln, OS)
#     pragma linkage (cmesl,  OS)
#     pragma linkage (cmesui, OS)
#     pragma linkage (cmetpn, OS)
#     pragma linkage (cmflus, OS)
#     pragma linkage (cminic, OS)
#     pragma linkage (cminit, OS)
#     pragma linkage (cmptr,  OS)
```

C Pseudonym File

```
#     pragma linkage (cmrcv,  OS)
#     pragma linkage (cmrltp, OS)
#     pragma linkage (cmrts,  OS)
#     pragma linkage (cmscsp, OS)
#     pragma linkage (cmscst, OS)
#     pragma linkage (cmscsu, OS)
#     pragma linkage (cmsct,  OS)
#     pragma linkage (cmsdt,  OS)
#     pragma linkage (cmsed,  OS)
#     pragma linkage (cmsend, OS)
#     pragma linkage (cmserr, OS)
#     pragma linkage (cmsf,   OS)
#     pragma linkage (cmsld,  OS)
#     pragma linkage (cmsltp, OS)
#     pragma linkage (cmsmn,  OS)
#     pragma linkage (cmspln, OS)
#     pragma linkage (cmspm , OS)
#     pragma linkage (cmsptr, OS)
#     pragma linkage (cmsrc,  OS)
#     pragma linkage (cmsrt,  OS)
#     pragma linkage (cmssl,  OS)
#     pragma linkage (cmsst,  OS)
#     pragma linkage (cmstpn, OS)
#     pragma linkage (cmtrts, OS)
#     pragma linkage (cmwait, OS)
#endif

#if defined(CM_OS400) || defined(CM_MVS)
#     pragma linkage (CMACCI, OS)
#     pragma linkage (CMACCP, OS)
#     pragma linkage (CMALLC, OS)
#     pragma linkage (CMCANC, OS)
#     pragma linkage (CMCFM,  OS)
#     pragma linkage (CMCFMD, OS)
#     pragma linkage (CMCNVI, OS)
#     pragma linkage (CMCNVO, OS)
#     pragma linkage (CMDEAL, OS)
#     pragma linkage (CMECS,  OS)
#     pragma linkage (CMECT,  OS)
#     pragma linkage (CMECTX, OS)
#     pragma linkage (CMEMBS, OS)
#     pragma linkage (CMEMN,  OS)
#     pragma linkage (CMEPLN, OS)
#     pragma linkage (CMESL,  OS)
#     pragma linkage (CMESUI, OS)
#     pragma linkage (CMETPN, OS)
#     pragma linkage (CMFLUS, OS)
#     pragma linkage (CMINIC, OS)
#     pragma linkage (CMINIT, OS)
#     pragma linkage (CMPTR,  OS)
#     pragma linkage (CMRCV,  OS)
#     pragma linkage (CMRLTP, OS)
#     pragma linkage (CMRTS,  OS)
#     pragma linkage (CMSCSP, OS)
#     pragma linkage (CMSCST, OS)
#     pragma linkage (CMSCSU, OS)
#     pragma linkage (CMSCT,  OS)
#     pragma linkage (CMSDT,  OS)
#     pragma linkage (CMSED,  OS)
#     pragma linkage (CMSEND, OS)
#     pragma linkage (CMSERR, OS)
#     pragma linkage (CMSF,   OS)
#     pragma linkage (CMSLD,  OS)
#     pragma linkage (CMSLTP, OS)
```

```
#     pragma linkage (CMSMN,  OS)
#     pragma linkage (CMSPLN, OS)
#     pragma linkage (CMSPM , OS)
#     pragma linkage (CMSPTR, OS)
#     pragma linkage (CMSRC,  OS)
#     pragma linkage (CMSRT,  OS)
#     pragma linkage (CMSSL,  OS)
#     pragma linkage (CMSST,  OS)
#     pragma linkage (CMSTPN, OS)
#     pragma linkage (CMTRTS, OS)
#     pragma linkage (CMWAIT, OS)
#endif

/*
 *    #pragma linkage directives - use on OS/2 32-bit only
 */
#if defined(ES32TO16)
#     pragma linkage (cmacci, far16 pascal)
#     pragma linkage (cmaccp, far16 pascal)
#     pragma linkage (cmallc, far16 pascal)
#     pragma linkage (cmcanc, far16 pascal)
#     pragma linkage (cmcfm,  far16 pascal)
#     pragma linkage (cmcfmd, far16 pascal)
#     pragma linkage (cmcnvi, far16 pascal)
#     pragma linkage (cmcnvo, far16 pascal)
#     pragma linkage (cmdeal, far16 pascal)
#     pragma linkage (cmecs,  far16 pascal)
#     pragma linkage (cmect,  far16 pascal)
#     pragma linkage (cmectx, far16 pascal)
#     pragma linkage (cmembs, far16 pascal)
#     pragma linkage (cmemn,  far16 pascal)
#     pragma linkage (cmepln, far16 pascal)
#     pragma linkage (cmesl,  far16 pascal)
#     pragma linkage (cmesui, far16 pascal)
#     pragma linkage (cmetpn, far16 pascal)
#     pragma linkage (cmflus, far16 pascal)
#     pragma linkage (cminic, far16 pascal)
#     pragma linkage (cminit, far16 pascal)
#     pragma linkage (cmptr,  far16 pascal)
#     pragma linkage (cmrcv,  far16 pascal)
#     pragma linkage (cmrltp, far16 pascal)
#     pragma linkage (cmrts,  far16 pascal)
#     pragma linkage (cmscsp, far16 pascal)
#     pragma linkage (cmscst, far16 pascal)
#     pragma linkage (cmscsu, far16 pascal)
#     pragma linkage (cmsct,  far16 pascal)
#     pragma linkage (cmsdt,  far16 pascal)
#     pragma linkage (cmsed,  far16 pascal)
#     pragma linkage (cmsend, far16 pascal)
#     pragma linkage (cmserr, far16 pascal)
#     pragma linkage (cmsf,   far16 pascal)
#     pragma linkage (cmsld,  far16 pascal)
#     pragma linkage (cmsltp, far16 pascal)
#     pragma linkage (cmsmn,  far16 pascal)
#     pragma linkage (cmspln, far16 pascal)
#     pragma linkage (cmspm , far16 pascal)
#     pragma linkage (cmsptr, far16 pascal)
#     pragma linkage (cmsrc,  far16 pascal)
#     pragma linkage (cmsrt,  far16 pascal)
#     pragma linkage (cmssl,  far16 pascal)
#     pragma linkage (cmsst,  far16 pascal)
#     pragma linkage (cmstpn, far16 pascal)
#     pragma linkage (cmtrts, far16 pascal)
#     pragma linkage (cmwait, far16 pascal)
```

C Pseudonym File

```
#     pragma linkage (xcecst, far16 pascal)
#     pragma linkage (xcecsu, far16 pascal)
#     pragma linkage (xcmdsi, far16 pascal)
#     pragma linkage (xcmesi, far16 pascal)
#     pragma linkage (xcmssi, far16 pascal)
#     pragma linkage (xcscsp, far16 pascal)
#     pragma linkage (xcscst, far16 pascal)
#     pragma linkage (xcscsu, far16 pascal)
#endif



#if defined(CM_OS2) || defined(CM_DOS) || defined(CM_AIX)

/*
 * Conversation security extensions
 */

typedef CM_INT32 XC_CONVERSATION_SECURITY_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

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

#if !defined(CM_DOS) /* extract calls are not supported in DOS */


/* Extract_Conversation_Security_Type */
CM_ENTRY xcecst(unsigned char  CM_PTR,   /* conversation_ID               */
                CM_INT32 CM_PTR,        /* conversation_security_type    */
                CM_RETURN_CODE 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_RETURN_CODE CM_PTR); /* return_code                   */
#endif

/* 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_RETURN_CODE 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_RETURN_CODE 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_RETURN_CODE CM_PTR); /* return_code                   */
```

```
#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif


#if defined(CM_OS2)
/*
 * Constants and prototypes for OS/2 side information calls
 */

typedef CM_INT32 XC_TP_NAME_TYPE;
/* 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;

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

/* Set_CPIC_Side_Information */
CM_ENTRY xcmssi(unsigned char CM_PTR,  /* key lock                          */
                SIDE_INFO CM_PTR,      /* side info_entry (see struct above)*/
                CM_INT32 CM_PTR,       /* side_info length                  */
                CM_RETURN_CODE 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 CM_PTR,      /* side_info_entry (see struct above)*/
                CM_INT32 CM_PTR,       /* side_info_length                  */
                CM_RETURN_CODE 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_RETURN_CODE CM_PTR);  /* return_code                     */
#ifdef __cplusplus
}
#endif /* __cplusplus */
#endif

#ifdef CM_VM

/*
```

C Pseudonym File

```
| * XC_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 XC_INT32 signed long int
|#define XC_ENTRY extern void
|#define XC_PTR *


|typedef XC_INT32 CMINT;
|typedef CMINT *PCMINT;
|typedef unsigned char CMCHAR;
|typedef CMCHAR *PCMCHAR;


|typedef XC_INT32 XC_RESOURCE_MANAGER_TYPE;
|/*  resource_manager_type values  */
|#define XC_PRIVATE                    (XC_RESOURCE_MANAGER_TYPE) 0
|#define XC_LOCAL                      (XC_RESOURCE_MANAGER_TYPE) 1
|#define XC_GLOBAL                     (XC_RESOURCE_MANAGER_TYPE) 2
|#define XC_SYSTEM                     (XC_RESOURCE_MANAGER_TYPE) 3

|typedef XC_INT32 XC_SERVICE_MODE;
|/*  service_mode values  */
|#define XC_SINGLE                     (XC_SERVICE_MODE) 0
|#define XC_SEQUENTIAL                 (XC_SERVICE_MODE) 1
|#define XC_MULTIPLE                   (XC_SERVICE_MODE) 2

|typedef XC_INT32 XC_SECURITY_LEVEL_FLAG;
|/*  security_level_flag values  */
|#define XC_REJECT_SECURITY_NONE       (XC_SECURITY_LEVEL_FLAG) 0
|#define XC_ACCEPT_SECURITY_NONE       (XC_SECURITY_LEVEL_FLAG) 1

|typedef XC_INT32 XC_CONVERSATION_SECURITY_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

|typedef XC_INT32 XC_EVENT_TYPE;
|/*  event_type values  */
|#define XC_ALLOCATION_REQUEST         (XC_EVENT_TYPE) 1
|#define XC_INFORMATION_INPUT          (XC_EVENT_TYPE) 2
|#define XC_RESOURCE_REVOKED           (XC_EVENT_TYPE) 3
|#define XC_CONSOLE_INPUT              (XC_EVENT_TYPE) 4
|#define XC_REQUEST_ID                 (XC_EVENT_TYPE) 5
|#define XC_USER_EVENT                 (XC_EVENT_TYPE) 6

|CM_ENTRY xcecl( unsigned char CM_PTR,   /* conversation_ID          */
|                unsigned char CM_PTR,   /* luwid                   */
|                CM_INT32 CM_PTR,        /* luwid_length            */
|                CM_RETURN_CODE CM_PTR); /* return_code             */
|CM_ENTRY xcecsu(unsigned char CM_PTR,   /* conversation_ID          */
|                unsigned char CM_PTR,   /* security_user_ID        */
|                CM_INT32 CM_PTR,        /* security_user_ID_length */
|                CM_RETURN_CODE CM_PTR); /* return_code             */
|CM_ENTRY xcecwu(unsigned char CM_PTR,   /* conversation_ID          */
|                CM_INT32 CM_PTR,        /* workunitid              */
|                CM_RETURN_CODE CM_PTR); /* return_code             */
|CM_ENTRY xcelfq(unsigned char CM_PTR,   /* conversation_ID          */
|                unsigned char CM_PTR,   /* local_fq_LU_name        */
|                CM_INT32 CM_PTR,        /* local_fq_LU_name_length */
|                CM_RETURN_CODE CM_PTR); /* return_code             */
```

```
CM_ENTRY xcerfq(unsigned char CM_PTR,   /* conversation_ID          */
                unsigned char CM_PTR,   /* remote_fq_LU_name       */
                CM_INT32 CM_PTR,        /* remote_fq_LU_name_length */
                CM_RETURN_CODE CM_PTR); /* return_code             */
CM_ENTRY xcetpn(unsigned char CM_PTR,   /* conversation_ID          */
                unsigned char CM_PTR,   /* TP_name                 */
                CM_INT32 CM_PTR,        /* TP_name_length          */
                CM_RETURN_CODE CM_PTR); /* return_code             */
CM_ENTRY xcidrm(unsigned char CM_PTR,   /* resource_ID             */
                CM_INT32 CM_PTR,        /* resource_manager_type   */
                CM_INT32 CM_PTR,        /* service_mode            */
                CM_INT32 CM_PTR,        /* security_level_flag     */
                CM_RETURN_CODE CM_PTR); /* return_code             */
CM_ENTRY xcscsp(unsigned char CM_PTR,   /* conversation_ID          */
                unsigned char CM_PTR,   /* security_password       */
                CM_INT32 CM_PTR,        /* security_password_length */
                CM_RETURN_CODE CM_PTR); /* return_code             */
CM_ENTRY xcscst(unsigned char CM_PTR,   /* conversation_ID          */
                CM_INT32 CM_PTR,        /* conv_security_type      */
                CM_RETURN_CODE CM_PTR); /* return_code             */
CM_ENTRY xcscsu(unsigned char CM_PTR,   /* conversation_ID          */
                unsigned char CM_PTR,   /* security_user_ID        */
                CM_INT32 CM_PTR,        /* security_user_ID_length  */
                CM_RETURN_CODE CM_PTR); /* return_code             */
CM_ENTRY xcscui(unsigned char CM_PTR,   /* conversation_ID          */
                unsigned char CM_PTR,   /* client_user_ID          */
                CM_RETURN_CODE CM_PTR); /* return_code             */
CM_ENTRY xcsue( unsigned char CM_PTR,   /* event_ID                */
                unsigned char CM_PTR,   /* user_data               */
                CM_INT32 CM_PTR,        /* user_data_length        */
                CM_RETURN_CODE CM_PTR); /* return_code             */
CM_ENTRY xctrrm(unsigned char CM_PTR,   /* resource_ID             */
                CM_RETURN_CODE CM_PTR); /* return_code             */
CM_ENTRY xcwoe( unsigned char CM_PTR,   /* resource_ID             */
                unsigned char CM_PTR,   /* conversation_ID         */
                CM_INT32 CM_PTR,        /* event_type              */
                CM_INT32 CM_PTR,        /* info_input_length       */
                unsigned char CM_PTR,   /* console_input_buffer    */
                CM_RETURN_CODE CM_PTR); /* return_code             */

#     pragma linkage (xcecl,  OS)
#     pragma linkage (xcecsu, OS)
#     pragma linkage (xcecwu, OS)
#     pragma linkage (xcelfq, OS)
#     pragma linkage (xcerfq, OS)
#     pragma linkage (xcetpn, OS)
#     pragma linkage (xcidrm, OS)
#     pragma linkage (xcscsp, OS)
#     pragma linkage (xcscst, OS)
#     pragma linkage (xcscsu, OS)
#     pragma linkage (xcscui, OS)
#     pragma linkage (xcsue,  OS)
#     pragma linkage (xctrrm, OS)
#     pragma linkage (xcwoe,  OS)

#endif
```

C Pseudonym File

```
/*
 * These macros allow you to write programs that are easier to read, since
 * you can use the full name of the CPI-C call rather than its 6-character
 * entry point.
 *
 * When porting code that uses these macros, you will have to ensure that
 * the macros are defined on the target platform.
 */

#ifdef READABLE_MACROS
#define Accept_Conversation(v1,v2)            cmaccp(v1,v2)
#define Accept_Incoming(v1, v2)               cmacci(v1,v2)
#define Allocate(v1,v2)                       cmallc(v1,v2)
#define Cancel_Conversation(v1,v2)            cmcanc(v1,v2)
#define Confirm(v1,v2,v3)                     cmcfm(v1,v2,v3)
#define Confirmed(v1,v2)                      cmcfmd(v1,v2)
#define Convert_Incoming(v1,v2,v3)            cmcnvi(v1,v2,v3)
#define Convert_Outgoing(v1,v2,v3)            cmcnvo(v1,v2,v3)
#define Deallocate(v1,v2)                     cmdeal(v1,v2)
#define Extract_Conversation_Context(v1,v2,v3,v4) cmectx(v1,v2,v3,v4)
#define Extract_Maximum_Buffer_Size(v1,v2)    cmembs(v1,v2)
#define Extract_Mode_Name(v1,v2,v3,v4)        cmemn(v1,v2,v3,v4)
#define Extract_Partner_LU_Name(v1,v2,v3,v4)  cmepln(v1,v2,v3,v4)
#define Extract_Security_User_ID(v1,v2,v3,v4) cmesui(v1,v2,v3,v4)
#define Extract_Sync_Level(v1,v2,v3)          cmesl(v1,v2,v3)
#define Extract_TP_Name(v1,v2,v3,v4)          cmetpn(v1,v2,v3,v4)
#define Flush(v1,v2)                          cmflus(v1,v2)
#define Initialize_Conversation(v1,v2,v3)     cminit(v1,v2,v3)
#define Initialize_For_Incoming(v1,v2)        cminic(v1,v2)
#define Prepare_To_Receive(v1,v2)             cmptr(v1,v2)
#define Receive(v1,v2,v3,v4,v5,v6,v7,v8)      cmrcv(v1,v2,v3,v4,v5,v6,v7,v8)
#define Release_Local_TP_Name(v1,v2,v3)       cmrltp(v1,v2,v3)
#define Request_To_Send(v1,v2)                cmrts(v1,v2)
#define Send_Data(v1,v2,v3,v4,v5)             cmsend(v1,v2,v3,v4,v5)
#define Send_Error(v1,v2,v3)                  cmserr(v1,v2,v3)
#define Set_Conversation_Security_Password(v1,v2,v3,v4) cmscsp(v1,v2,v3,v4)
#define Set_Conversation_Security_Type(v1,v2,v3) cmscst(v1,v2,v3)
#define Set_Conversation_Security_User_ID(v1,v2,v3,v4) cmscsu(v1,v2,v3,v4)
#define Set_Conversation_Type(v1,v2,v3)       cmsct(v1,v2,v3)
#define Set_Deallocate_Type(v1,v2,v3)         cmsdt(v1,v2,v3)
#define Set_Error_Direction(v1,v2,v3)         cmsed(v1,v2,v3)
#define Set_Fill(v1,v2,v3)                    cmsf(v1,v2,v3)
#define Set_Log_Data(v1,v2,v3,v4)             cmsld(v1,v2,v3,v4)
#define Set_Mode_Name(v1,v2,v3,v4)            cmsmn(v1,v2,v3,v4)
#define Set_Partner_LU_Name(v1,v2,v3,v4)      cmspln(v1,v2,v3,v4)
#define Set_Prepare_To_Receive_Type(v1,v2,v3) cmsptr(v1,v2,v3)
#define Set_Processing_Mode(v1,v2,v3)         cmspm(v1,v2,v3)
#define Set_Processing_Mode(v1,v2,v3)         cmspm(v1,v2,v3)
#define Set_Receive_Type(v1,v2,v3)            cmsrt(v1,v2,v3)
#define Set_Return_Control(v1,v2,v3)          cmsrc(v1,v2,v3)
#define Set_Send_Type(v1,v2,v3)               cmsst(v1,v2,v3)
#define Set_Sync_Level(v1,v2,v3)              cmssl(v1,v2,v3)
#define Set_TP_Name(v1,v2,v3,v4)              cmstpn(v1,v2,v3,v4)
#define Specify_Local_TP_Name(v1,v2,v3)       cmsltp(v1,v2,v3)
#define Test_Request_To_Send_Received(v1,v2,v3) cmtrts(v1,v2,v3)
#define Wait_For_Conversation(v1,v2,v3)       cmwait(v1,v2,v3)

#endif


#endif
```

COBOL Pseudonym File (CMCOBOL)

```
*Edition SC31-6180-00
*
*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.
     88  CM-INITIALIZE-INCOMING-STATE     VALUE 14.
*
 01  CONVERSATION-TYPE          PIC 9(9) COMP-4.
     88  CM-BASIC-CONVERSATION    VALUE 0.
     88  CM-MAPPED-CONVERSATION   VALUE 1.
*
 01  CONVERSATION-SECURITY-TYPE PIC 9(9) COMP-4.
     88  CM-SECURITY-NONE                 VALUE 0.
     88  CM-SECURITY-SAME                 VALUE 1.
     88  CM-SECURITY-PROGRAM              VALUE 2.
*
 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-OPERATION-INCOMPLETE          VALUE 35.
                    88   CM-SYSTEM-EVENT                  VALUE 36.
                    88   CM-OPERATION-NOT-ACCEPTED        VALUE 37.
                    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  CONVERSATION_RETURN_CODE       PIC 9(9) COMP-4.
                    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-OPERATION-INCOMPLETE          VALUE 35.
                    88   CM-SYSTEM-EVENT                  VALUE 36.
                    88   CM-OPERATION-NOT-ACCEPTED        VALUE 37.
                    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  PROCESSING-MODE            PIC 9(9) COMP-4.
     88   CM-BLOCKING           VALUE 0.
     88   CM-NON-BLOCKING       VALUE 1.
*
 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.
*
```

COBOL Pseudonym File

```
 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*************** CPI Communications Variable Names *********************
C*
C*                      Edition SC31-6180-00
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 CM_INITIALIZE_INCOMING_STATE   /14/

        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/
        INTEGER INCOME                         /14/

C *** conversation_type ****************************************
C
        INTEGER CM_BASIC_CONVERSATION  /0/
        INTEGER CM_MAPPED_CONVERSATION /1/

        INTEGER BASIC                  /0/
        INTEGER MAPPED                 /1/

C *** conversation_security_type *******************************
C
        INTEGER CM_SECURITY_NONE       /0/
        INTEGER CM_SECURITY_SAME       /1/
        INTEGER CM_SECURITY_PROGRAM    /2/

        INTEGER NONE                   /0/
        INTEGER SAME                   /1/
        INTEGER PROGRAM                /2/
```

FORTRAN Pseudonym File

```
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 *** processing_mode ****************************************
C
        INTEGER CM_BLOCKING          /0/
        INTEGER CM_NON_BLOCKING      /1/

        INTEGER BLOCK                /0/
        INTEGER NONBLK               /1/
```

```
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_OPERATION_INCOMPLETE         /35/
|       INTEGER CM_SYSTEM_EVENT                 /36/
|       INTEGER CM_OPERATION_NOT_ACCEPTED       /37/
        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/
```

FORTRAN Pseudonym File

```
      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 OPINC                         /35/
      INTEGER SYSEVT                        /36/
      INTEGER OPNACC                        /37/
      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 CMACCI
        EXTERNAL CMACCP
        EXTERNAL CMALLC
        EXTERNAL CMCANC
        EXTERNAL CMCFM
        EXTERNAL CMCFMD
        EXTERNAL CMCNVI
        EXTERNAL CMCNVO
        EXTERNAL CMDEAL
        EXTERNAL CMECS
        EXTERNAL CMECT
        EXTERNAL CMECTX
        EXTERNAL CMEMBS
        EXTERNAL CMEMN
        EXTERNAL CMEPLN
        EXTERNAL CMESL
        EXTERNAL CMESUI
        EXTERNAL CMETPN
        EXTERNAL CMFLUS
        EXTERNAL CMINIC
        EXTERNAL CMINIT
        EXTERNAL CMPTR
        EXTERNAL CMRCV
        EXTERNAL CMRLTP
        EXTERNAL CMRTS
```

FORTRAN Pseudonym File

```
|                               EXTERNAL CMSCSP
|                               EXTERNAL CMSCST
|                               EXTERNAL CMSCSU
                                EXTERNAL CMSCT
                                EXTERNAL CMSDT
                                EXTERNAL CMSED
                                EXTERNAL CMSEND
                                EXTERNAL CMSERR
                                EXTERNAL CMSF
                                EXTERNAL CMSLD
|                               EXTERNAL CMSLTP
                                EXTERNAL CMSMN
                                EXTERNAL CMSPLN
|                               EXTERNAL CMSPM
                                EXTERNAL CMSPTR
                                EXTERNAL CMSRC
                                EXTERNAL CMSRT
                                EXTERNAL CMSSL
                                EXTERNAL CMSST
                                EXTERNAL CMSTPN
                                EXTERNAL CMTRTS
|                               EXTERNAL CMWAIT
                      C************* End of CPI Communications routine prototypes ***********
```

PL/I Pseudonym File (CMPLI)

```
/*********************************************************************/
/*                                                                   */
/* PL/I INCLUDE FILE FOR CPI COMMUNICATIONS SUPPORT                  */
/*                                                                   */
/*               Edition SC31-6180-00                                */
/*                                                                   */
/*********************************************************************/
%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))
    cm_initialize_incoming_state    INITIAL(14))
                                    FIXED BINARY(31) STATIC;
%skip;
/*********************************************************************/
/* Conversation Type Values                                          */
/*********************************************************************/
DECLARE
  ( cm_basic_conversation           INITIAL(0),
    cm_mapped_conversation          INITIAL(1))
                                    FIXED BINARY(31) STATIC;
%skip;
/*********************************************************************/
/* Conversation Security Type Values                                 */
/*********************************************************************/
DECLARE
  ( cm_security_none                INITIAL(0),
    cm_security_same                INITIAL(1),
    cm_security_program             INITIAL(2))
                                    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;
  /*****************************************************************/
  /* Processing Mode Values                                        */
  /*****************************************************************/
  DECLARE
    ( cm_blocking                       INITIAL(0),
      cm_non_blocking                   INITIAL(1))
                                        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 and Conversation_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_operation_incomplete          INITIAL(35),
|     cm_system_event                  INITIAL(36),
|     cm_operation_not_accepted        INITIAL(37),
      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;
  /*****************************************************************/
  /* CPI Communications Accept Incoming Call                       */
  /*****************************************************************/
  DECLARE
    CMACCI ENTRY
      ( CHAR(8),                   /* In  - Conversation ID         */
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /*****************************************************************/
  /* CPI Communications Accept Conversation Call                   */
  /*****************************************************************/
  DECLARE
    CMACCP ENTRY
      ( CHAR(8),                   /* Out - Conversation ID         */
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /*****************************************************************/
  /* CPI Communications Allocate Call                              */
  /*****************************************************************/
  DECLARE
    CMALLC ENTRY
      ( CHAR(8),                   /* In  - Conversation ID         */
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /*****************************************************************/
  /* CPI Communications Cancel Call                                */
  /*****************************************************************/
  DECLARE
    CMCANC ENTRY
      ( CHAR(8),                   /* In  - Conversation ID         */
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /*****************************************************************/
  /* 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;
  /*****************************************************************/
  /* CPI Communications Confirmed Call                             */
  /*****************************************************************/
  DECLARE
    CMCFMD ENTRY
      ( CHAR(8),                   /* In  - Conversation ID         */
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
```

```
%skip;
  /*****************************************************************/
  /* CPI Communications Convert Incoming Call                      */
  /*****************************************************************/
  DECLARE
    CMCNVI ENTRY
      ( CHAR(*),                  /* In/Out - Buffer               */
        FIXED BINARY(31),         /* In  - Buffer Length           */
        FIXED BINARY(31) )        /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /*****************************************************************/
  /* CPI Communications Convert Outgoing Call                      */
  /*****************************************************************/
  DECLARE
    CMCNVO ENTRY
      ( CHAR(*),                  /* In/Out - Buffer               */
        FIXED BINARY(31),         /* In  - Buffer Length           */
        FIXED BINARY(31) )        /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /*****************************************************************/
  /* CPI Communications Deallocate Call                            */
  /*****************************************************************/
  DECLARE
    CMDEAL ENTRY
      ( CHAR(8),                  /* In  - Conversation ID         */
        FIXED BINARY(31) )        /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /*****************************************************************/
  /* 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;
  /*****************************************************************/
  /* 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;
  /*****************************************************************/
  /* CPI Communications Extract_Conversation_Context               */
  /*****************************************************************/
  DECLARE
    CMECTX ENTRY
      ( CHAR(8),                  /* In  - Conversation ID         */
        CHAR(32),                 /* Out - Context                 */
        FIXED BINARY(31),         /* Out - Context Length          */
        FIXED BINARY(31) )        /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
```

```
%skip;
  /*****************************************************************/
  /* CPI Communications Extract_Maximum_Buffer_Size                */
  /*****************************************************************/
  DECLARE
    CMEMBS ENTRY
      ( FIXED BINARY(31),          /* Out - Maximum Buffer Size     */
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /*****************************************************************/
  /* 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;
  /*****************************************************************/
  /* 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;
  /*****************************************************************/
  /* 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;
  /*****************************************************************/
  /* CPI Communications Extract_Security_User_ID                   */
  /*****************************************************************/
  DECLARE
    CMESUI ENTRY
      ( CHAR(8),                   /* In  - Conversation ID         */
        CHAR(10),                  /* Out - Security User ID        */
        FIXED BINARY(31),          /* Out - Security User ID Length */
        FIXED BINARY(31) )         /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
```

```
%skip;
  /*****************************************************************/
  /* CPI Communications Extract_TP_Name                            */
  /*****************************************************************/
  DECLARE
    CMETPN ENTRY
      ( CHAR(8),                  /* In  - Conversation ID         */
        CHAR(64),                 /* Out - TP_Name                 */
        FIXED BINARY(31),         /* Out - TP_Name Length          */
        FIXED BINARY(31) )        /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /*****************************************************************/
  /* CPI Communications Flush Call                                 */
  /*****************************************************************/
  DECLARE
    CMFLUS ENTRY
      ( CHAR(8),                  /* In  - Conversation ID         */
        FIXED BINARY(31) )        /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /*****************************************************************/
  /* CPI Communications Initialize for Incoming                    */
  /*****************************************************************/
  DECLARE
    CMINIC ENTRY
      ( CHAR(8),                  /* Out - Conversation ID         */
        FIXED BINARY(31) )        /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /*****************************************************************/
  /* 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;
  /*****************************************************************/
  /* 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;
  /*****************************************************************/
  /* 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);
```

PL/I Pseudonym File

```
%skip;
  /*****************************************************************/
  /* CPI Communications Release Local TP Name                      */
  /*****************************************************************/
  DECLARE
    CMRLTP ENTRY
      ( CHAR(64,                  /* In  - TP_Name                 */
        FIXED BINARY(31),         /* In  - TP_Name_Length          */
        FIXED BINARY(31) )        /* In  - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /*****************************************************************/
  /* 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;
  /*****************************************************************/
  /* 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;
  /*****************************************************************/
  /* 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;
  /*****************************************************************/
  /* 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;
  /*****************************************************************/
  /* CPI Communications Set Conversation Security Password         */
  /*****************************************************************/
  DECLARE
    CMSCSP ENTRY
      ( CHAR(8),                  /* In  - Conversation ID         */
        CHAR(10),                 /* In  - Security Password       */
        FIXED BINARY(31),         /* In  - Secur. Password Length  */
        FIXED BINARY(31) )        /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
```

```
%skip;
  /*****************************************************************/
  /* CPI Communications Set Conversation Security Type             */
  /*****************************************************************/
  DECLARE
    CMSCST ENTRY
      ( CHAR(8),                  /* In  - Conversation ID         */
        FIXED BINARY(31),         /* In  - Security Type           */
        FIXED BINARY(31) )        /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /*****************************************************************/
  /* CPI Communications Set Conversation Security User ID          */
  /*****************************************************************/
  DECLARE
    CMSCSU ENTRY
      ( CHAR(8),                  /* In  - Conversation ID         */
        CHAR(10),                 /* In  - Security User ID        */
        FIXED BINARY(31),         /* In  - Security User ID        */
        FIXED BINARY(31) )        /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /*****************************************************************/
  /* 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;
  /*****************************************************************/
  /* 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;
  /*****************************************************************/
  /* 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;
  /*****************************************************************/
  /* 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);
```

PL/I Pseudonym File

```
%skip;
  /*****************************************************************/
  /* 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;
  /*****************************************************************/
  /* 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;
  /*****************************************************************/
  /* 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;
  /*****************************************************************/
  /* CPI Communications Set Processing Mode Call                   */
  /*****************************************************************/
  DECLARE
    CMSPM ENTRY
      ( CHAR(8),                  /* In  - Conversation ID         */
        FIXED BINARY(31),         /* In  - Processing Mode         */
        FIXED BINARY(31) )        /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
%skip;
  /*****************************************************************/
  /* 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;
  /*****************************************************************/
  /* 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;
  /*****************************************************************/
  /* 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;
  /*****************************************************************/
  /* 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;
  /*****************************************************************/
  /* 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;
  /*****************************************************************/
  /* CPI Communications Specify Local TP Name                      */
  /*****************************************************************/
  DECLARE
    CMSLTP ENTRY
      ( CHAR(64)                  /* In  - TP Name                 */
        FIXED BINARY(31),         /* In  - TP Name Length          */
        FIXED BINARY(31) )        /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
```

PL/I Pseudonym File

```
%skip;
  /*****************************************************************/
  /* 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);
%skip;
  /*****************************************************************/
  /* CPI Communications Wait For Conversation Call                 */
  /*****************************************************************/
  DECLARE
    CMWAIT ENTRY
      ( CHAR(8),                  /* Out - Conversation ID         */
        FIXED BINARY(31),         /* Out - Conversation Return Code*/
        FIXED BINARY(31) )        /* Out - Return Code             */
      EXTERNAL OPTIONS(ASM INTER);
```

REXX Pseudonym File (CMREXX)

```
                                                    /* Edition SC31-6180-00        */
                                                    /* 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
                CM_INITIALIZE_INCOMING_STATE   = 14
                                                    /* conversation_type           */
                CM_BASIC_CONVERSATION          = 0
                CM_MAPPED_CONVERSATION         = 1
                                                    /* conversation_security_type  */
                CM_SECURITY_NONE               = 0
                CM_SECURITY_SAME               = 1
                CM_SECURITY_PROGRAM            = 2
                                                    /* 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
                                                    /* processing_mode             */
                CM_BLOCKING                    = 0
                CM_NON_BLOCKING                = 1
                                                    /* 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
```

REXX Pseudonym File

```
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_OPERATION_INCOMPLETE         = 35
CM_SYSTEM_EVENT                 = 36
CM_OPERATION_NOT_ACCEPTED       = 37
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* Edition SC31-6180-00
I*
I* RPG INCLUDE FOR CPI 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*   CM_INITIALIZE_INCOMING_STATE     -- VALUE 14  (INCOM)
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              14                    C         INCOM
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* conversation_security_type values:
I*
I*   CM_SECURITY_NONE                 -- VALUE 0   (NONE)
I*   CM_SECURITY_SAME                 -- VALUE 1   (SAME)
I*   CM_SECURITY_PROGRAM              -- VALUE 2   (PROGRM)
I*
I              0                     C         NONE
I              1                     C         SAME
I              2                     C         PROGRM
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* processing_mode values:
I*
I*   CM_BLOCKING                      -- VALUE 0   (BLOCK)
I*   CM_NON_BLOCKING                  -- VALUE 1   (NBLOCK)
I*
I              0                     C         BLOCK
I              1                     C         NBLOCK
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 and conversation_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_OPERATION_INCOMPLETE         -- VALUE 35  (OPINC)
I*   CM_SYSTEM_EVENT                 -- VALUE 36  (SYSEVT)
I*   CM_OPERATION_NOT_ACCEPTED       -- VALUE 37  (OPNACC)
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
```

RPG Pseudonym File

```
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            35                    C         OPINC
|I            36                    C         SYSEVT
|I            37                    C         OPNACC
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
```

RPG Pseudonym File

Appendix F. Sample Programs

This appendix contains the following sections:

- "SALESRPT (Initiator of the Conversation)" on page 298

  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 302

  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 307

  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. 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 of error handling will depend on the nature of actual applications.

© Copyright IBM Corp. 1988, 1993

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
```

Results

Summary of Changes

This book is based on the seventh edition of the *SAA Common Programming Interface: Communications Reference*, SC26-4399-06. Revision bars are used in this book to show information that is new or changed from SC26-4399-06. The major new and changed topics are:

- Several new calls have been added to perform these functions:

  Functions Calls

  Non-blocking
  Set_Processing_Mode
  Wait_For_Conversation
  Cancel_Conversation

  Accept Multiple
  Specify_Local_TP_Name
  Release_Local_TP_Name
  Extract_TP_Name
  Initialize_For_Incoming
  Accept_Incoming

  Security
  Set_Conversation_Security_Type
  Set_Conversation_Security_User_ID
  Set_Conversation_Security_Password
  Extract_Security_User_ID

  Data conversion utilities
  Convert_Incoming
  Convert_Outgoing

  Extended buffer size
  Extract_Maximum_Buffer_Size

  Context management
  Extract_Conversation_Context

- Several new sections have been added to Chapter 2, “CPI Communications Terms and Concepts” on page 7.
- Two new flow examples have been added to Chapter 3, “Program-to-Program Communication Tutorial” on page 35.
- Appendix B, “Return Codes” on page 201 has been updated.
- Appendix C, “State Table” on page 211 has been updated.
- Appendix D, “CPI Communications and LU 6.2” on page 225 has been updated.
- Appendix E, “Pseudonym Files” on page 231 has been updated.

© Copyright IBM Corp. 1988, 1993

Glossary

A

alias. (1) An alternative name used to identify an object, a logical unit, or a database. (2) A nickname set up by the network administrator for a file, printer, or serial device.

API. application programming interface.

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.

blocking. A CPI Communications call-processing mode in which a call operation completes before control is returned to the program. The program is blocked (unable to perform any other work) until the call operation is completed.

C

context. A system-wide entity used by node services to group logical attributes for work done by a program on behalf of a partner program.

context identifier. A system-wide identifier used by node services to identify a context.

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.

current context. For a program, the context within which work is currently being done.

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.

N

network name. In SNA, the symbolic identifier by which end users refer to a network accessible unit (NAU), link station, or link.

non-blocking. A CPI Communications call-processing mode in which, if possible, a call operation completes immediately. If the call operation cannot complete immediately, control is returned to the program with the CM_OPERATION_INCOMPLETE return code. The call operation remains in progress, and completion of the call operation occurs at a later time. Meanwhile, the program is free to perform other work.

O

outstanding operation. A call operation for which the program has received the CM_OPERATION_INCOMPLETE return code. The call remains in progress, and completion occurs at a later time. An outstanding operation can only occur on a conversation using non-blocking processing mode.

© Copyright IBM Corp. 1988, 1993

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 a resource recovery interface and a sync point manager.

pseudonym file. A file that provides CPI Communications declarations for a particular programming language.

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. An 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 *conversation_security_type*, *mode_name*, *partner_LU_name*, *security_password*, *security_user_ID*, 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 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

Special Characters

_ (underscore) 26, 67
. (period) 199
- (dash) 67
& (ampersand) 67

A

abnormal program ending 12
Accept Conversation (CMACCP)
  call description 73
  example flow using 39
Accept_Incoming (CMACCI)
  call description 75
  example flow using 60
advanced function calls
  description 13
  examples 43—59
  list 14
advanced program-to-program communications
  See also LU 6.2
  type 6.2 logical unit 8
  verbs 228
Allocate (CMALLC)
  call description 77
  example flow using 39
APPC
  See advanced program-to-program communications
Application Generator considerations 67
ASCII, conversion of 199

B

backout-required condition
  described 30
  effect on multiple conversations 25
basic conversation 9, 140
begin conversation
  CMALLC (Allocate) 77
  example flow 37
  program startup 12
  simple example 18
blank *sym_dest_name* 11, 115
blocking operations 22
buffering of data
  description 43, 140
  example flow 38

C

C considerations 67
calls
  advanced function
    examples 43—59
    list 14
  description 13
  for resource recovery interfaces 28
  naming conventions 26
  starter-set
    examples 36—41
    list 14
  table of 13, 70
Cancel_Conversation (CMCANC) 80
changing data flow direction
  by receiving program 48
  by sending program 40, 44
character sets
  exceptions for SNA TP names 227
  general 195
character strings 197
characteristics
  automatic conversion of 22
  comparison of defaults 20
  context_ID, extract 96
  conversation_security_type
    possible values 192
    set 151
  conversation_state
    extract 98
    possible values 192
  conversation_type
    extract 100
    possible values 192
    set 155
  deallocate_type
    possible values 192
    set 157
  default values 15
  described 18
  error_direction
    possible values 192
    set 160
  fill
    possible values 192
    set 162
  how to examine 19
  initial values, table of 20
  integer values 191
  log_data_length, set 164
  log_data, set 164
  mode_name
    extract 103
    set 166

© Copyright IBM Corp. 1988, 1993

characteristics *(continued)*
mode_name_length
extract 103
set 166
modifying 19
naming conventions 26
overview 18
partner_LU_name
extract 105
set 168
partner_LU_name_length
extract 105
set 168
prepare_to_receive_type
possible values 192
set 170
processing_mode
possible values 192
set 172
pseudonyms 26
receive_type
possible values 192
set 174
return_control
possible values 192
set 176
security_password_length, set 149
security_password, set 149
security_user_ID
extract 107
set 153
security_user_ID_length, set 153
send_type
possible values 192
set 178
sync_level
extract 109
possible values 192
set 180
TP_name
extract 111
set 182
TP_name_length
set 182
viewing 19
CMACCI (Accept_Incoming)
call description 75
example flow using 60
CMACCP (Accept_Conversation)
call description 73
example flow using 39
CMALLC (Allocate)
call description 77
example flow using 39
CMCANC (Cancel_Conversation) 80
CMCFM (Confirm)
call description 82
example flow using 47
CMCFMD (Confirmed)
call description 85
example flow using 47
CMCNVI (Convert_Incoming) 87
CMCNVO (Convert_Outgoing) 89
CMDEAL (Deallocate)
call description 91
example flow using 39
CMECS (Extract_Conversation_State) 98
CMECT (Extract_Conversation_Type) 100
CMECTX (Extract_Conversation_Context) 96
CMEMBS (Extract_Maximum_Buffer_Size) 102
CMEMN (Extract_Mode_Name) 103
CMEPLN (Extract_Partner_LU_Name) 105
CMESL (Extract_Sync_Level) 109
CMESUI (Extract_Security_User_ID) 107
CMETPN (Extract_TP_Name) 111
CMFLUS (Flush)
call description 113
example flow using 47
CMINIC (Initialize_For_Incoming)
call description 117
example flow using 60
CMINIT (Initialize_Conversation)
call description 115
example flow using 39
CMPTR (Prepare_To_Receive)
call description 119
example flow using 45, 49
CMRCV (Receive)
call description 123
example flow using 39
CMRLTP (Release_Local_TP_Name) 133
CMRTS (Request_To_Send)
call description 135
example flow using 49
CMSCSP (Set_Conversation_Security_Password)
call description 149
CMSCST (Set_Conversation_Security_Type)
call description 151
CMSCSU (Set_Conversation_Security_User_ID)
call description 153
CMSCT (Set_Conversation_Type) 155
CMSDT (Set_Deallocate_Type) 157
CMSED (Set_Error_Direction) 160
CMSEND (Send_Data)
call description 137
example flow using 39
CMSERR (Send_Error)
call description 143
example flow using 51
CMSF (Set_Fill) 162

CMSLD (Set_Log_Data)
call description 164
CMSLTP (Specify_Local_TP_Name) 184
CMSMN (Set_Mode_Name)
call description 166
CMSPLN (Set_Partner_LU_Name)
call description 168
CMSPM (Set_Processing_Mode) 172
CMSPTR (Set_Prepare_To_Receive_Type) 170
CMSRC (Set_Return_Control) 176
CMSRT (Set_Receive_Type) 174
CMSSL (Set_Sync_Level)
call description 180
example flow using 47
CMSST (Set_Send_Type)
call description 178
example flow using 49
CMSTPN (Set_TP_Name)
call description 182
CMTRTS (Test_Request_To_Send_Received) 185
CMWAIT (Wait_For_Conversation)
call description 187
example flow using 62
COBOL considerations 67
common programming interface (CPI)
communications
See CPI Communications
communication
across an SNA network 8
with an APPC program 227
concurrent conversations 15
Confirm (CMCFM)
call description 82
example flow using 47
Confirm state 25
Confirm-Deallocate state 25
Confirm-Send state 25
confirmation processing
Confirm call 82
Confirmed call 85
example flow 47
Confirmed (CMCFMD)
call description 85
example flow using 47
contexts
and conversations 17
and security parameters 17
changing 17
creating 17
current 17
setting 17
conventions, naming 26
conversation
accept 73
allocate 77
basic 9, 140
conversation *(continued)*
canceling 23, 80
characteristics
See characteristics
concurrent 15
dangling
See dangling conversation, deallocating
deallocate 91
description 9
examples 18, 36—59
identifier 14
initialize 115
mapped 9, 140
multiple 15
security 24
startup request 12, 18, 37
states
See states, conversation
synchronization and control
Confirm call 82
Confirmed call 85
Flush call 113
Prepare_To_Receive call 119
Request_To_Send call 135
Send_Error 143
Test_Request_To_Send_Received 185
transition from a state 25
types 9
conversation_ID
described 14
conversation security 24
conversation states
description 25
extracting 98
list 25
possible values 192
pseudonyms 26
table 219
conversation_type
extract 100
possible values 192
set 155
conversation_return_code
described 187
possible values 187, 192
conversion
of characteristics 22
of data 24
conversion to and from ASCII 199
Convert_Incoming (CMCNVI) 87
Convert_Outgoing (CMCNVO) 89
CPI Communications
communication with APPC programs 227
in SNA networks 8
naming conventions 26
overview 7

CPI Communications *(continued)*
program operating environment 10
relationship to LU 6.2 interface 225
versions of, table 4
with resource recovery interfaces 28
CPI-C
See CPI Communications
CSP considerations 67

D

dangling conversation, deallocating 12
data
buffering and transmission 43
direction, changing
by receiving program 48
by sending program 40, 44
flow
in both directions 40
in one direction 37
purging 50, 147
reception and validation of 46
data records
description 9
Receive call 124
Send_Data call 140
data_received parameter 13
Deallocate (CMDEAL)
call description 91
example flow using 39
deallocate_type characteristic
possible values 192
set 157
destination name, symbolic
blank 11, 115
defined 11
example 37

E

EBCDIC, conversion to
automatic 22
error_direction characteristic
and Send-Pending state 52, 226
possible values 192
set 160
error reporting
example 50
Send_Error call 146
examining conversation characteristics 19
See also extract calls
extract calls
Conversation_Context (CMECTX) 96
Conversation_State (CMECS) 98
Conversation_Type (CMECT) 100
Maximum_Buffer_Size (CMEMBS) 102
extract calls *(continued)*
Mode_Name (CMEMN) 103
Partner_LU_Name (CMEPLN) 105
Security_User_ID (CMESUI) 107
Sync_Level (CMESL) 109
TP_Name (CMETPN) 111
Extract_Conversation_Context (CMECTX) 96
Extract_Conversation_State (CMECS) 98
Extract_Conversation_Type (CMECT) 100
Extract_Maximum_Buffer_Size (CMEMBS) 102
Extract_Mode_Name (CMEMN) 103
Extract_Partner_LU_Name (CMEPLN) 105
Extract_Security_User_ID (CMESUI) 107
Extract_Sync_Level (CMESL) 109
Extract_TP_Name (CMETPN) 111

F

fill characteristic
possible values 192
set 162
flow
definition of 35
diagrams 37—59
Flush (CMFLUS)
call description 113
example flow using 47
format of calls 65, 66
FORTRAN considerations 67

G

graphic representations for character sets
table showing 195

I

initialize
conversation 115
state 25
Initialize_Conversation (CMINIT)
call description 115
example flow using 39
Initialize_For_Incoming (CMINIC)
call description 117
example flow using 60
Initialize_Incoming state 26
integer values 191
interface, communications
See CPI Communications

K

key topics 69

L

language considerations, programming 67
local partner 9
log_data characteristic
  set 164
logical records
  description 9
  Receive call 124
  Send_Data call 140
logical unit
  See also LU 6.2
  illustration 8
LU
  See logical unit
LU 6.2
  and CPI Communications 225
  application programming interface 225—230
  verbs 228

M

MAP_NAME 225
mapped conversation 9, 140
mode name, defined 11
mode_name characteristic
  defined 11
  extract 103
  length 199
  set 166
mode_name SNASVCMG
  Allocate call 77
  Set_Mode_Name call 166
mode, processing 22, 172
modifying conversation characteristics 19
  See also set calls
multiple conversations 15

N

naming conventions 26
node services 12
non-blocking
  and context management 23
  calls 23
  operations 22

O

outstanding operations 23, 187

P

parameters
  input 13
  output 13
partner 9
partner_LU_name characteristic
  extract 105
  length 199
  set 168
PIP data 225
PL/I considerations 68
Prepare_To_Receive (CMPTR)
  call description 119
  example flow using 45, 49
prepare_to_receive_type characteristic
  possible values 192
  set 170
program
  calls 12
  compilation 12
  partners 9
  startup processing 12
  states
    See states, conversation
  termination processing 12
programming language considerations 67
protected conversation 28
protected resource 28
pseudonym
  example of 27
  explanation of 26
  values 191
pseudonym files
  Application Generator (CSP) 232
  C 250
  COBOL 267
  FORTRAN 271
  PL/I 277
  REXX 289
  RPG 291

R

Receive (CMRCV)
  call description 123
  example flow using 39
Receive state
  description 25
  how a program enters 135
receive_type characteristic
  possible values 192
  set 174
records, logical
  description 9
  Receive call 124
  Send_Data call 140
Release_Local_TP_Name (CMRLTP) 133
remote partner
  definition 9
  residing on local system 9

reporting errors
  example 50
  Send_Error call 146
Request_To_Send (CMRTS)
  call description 135
  example flow using 49
request_to_send_received parameter 13
Reset state 25
resource recovery interfaces
  described 28
return codes 192, 201—210
return_code parameter
  definitions of values 201—210
  described 13
  possible values 192
return_control characteristic
  possible values 192
  set 176
REXX considerations 68

S

SAA resource recovery interface
  with CPI Communications 28
sample programs
  CREDRPT program 302
  introduction 297
  pseudonym file for 267
  results of 307
  SALESRPT program 298
Send state 25
Send-Pending state
  and *error_direction* characteristic 226
  description 25
Send_Data (CMSEND)
  call description 137
  example flow using 39
Send_Error (CMSERR)
  call description 143
  example flow using 51
send_type characteristic
  possible values 192
  set 178
service transaction programs 227
session 8
set calls
  conversation_security_type 151
  conversation_type 155
  deallocate_type 157
  error_direction 160
  fill 162
  log_data 164
  mode_name 166
  partner_LU_name 168
  prepare_to_receive_type 170
  processing_mode 172
set calls *(continued)*
  receive_type 174
  return_control 176
  security_password 149
  security_user_ID 153
  send_type 178
  sync_level 180
  TP_name 182
Set_Conversation_Security_Password (CMSCSP)
  call description 149
Set_Conversation_Security_Type (CMSCST)
  call description 151
Set_Conversation_Security_User_ID (CMSCSU)
  call description 153
Set_Conversation_Type (CMSCT) 155
Set_Deallocate_Type (CMSDT) 157
Set_Error_Direction (CMSED) 160
Set_Fill (CMSF) 162
Set_Log_Data (CMSLD)
  call description 164
Set_Mode_Name (CMSMN)
  call description 166
Set_Partner_LU_Name (CMSPLN)
  call description 168
Set_Prepare_To_Receive_Type (CMSPTR) 170
Set_Processing_Mode (CMSPM) 172
Set_Receive_Type (CMSRT) 174
Set_Return_Control (CMSRC) 176
Set_Send_Type (CMSST)
  call description 178
  example flow using 49
Set_Sync_Level (CMSSL)
  call description 180
  example flow using 47
Set_TP_Name (CMSTPN)
  call description 182
side information, defining
  overview 12
  purpose 11
  setting and accessing 12
SNA
  network 8
  service transaction programs 227
Specify_Local_TP_Name (CMSLTP) 184
starter-set calls
  description 13
  examples 36—41
  list 14
state table for conversations 219—223
  abbreviations 211
  example of how to use 217
state transition 25
states, conversation
  description 25
  extracting 98
  list 25

states, conversation *(continued)*
possible values 192
pseudonyms 26
table 219
status_received parameter 13
strings, character 197
sym_dest_name
See symbolic destination name
symbolic destination name
blank 11, 115
defined 11
example 37
sync point
described 28
sync_level characteristic
extract 109
possible values 192
set 180
synchronization point
See sync point
Systems Network Architecture
network 8
service transaction programs 227

T

take-backout notification 29
take-commit notification 29
Test_Request_To_Send_Received (CMTRTS) 185
TP_name characteristic
extract 111
set 182
transition, state 25
transmission of data 43
tutorial information
example flows 35—59
how to use this book 2
terms and concepts 7—33
types of conversations 9

U

USER_CONTROL_DATA 225

V

validation of data reception 46
values
integers 191
pseudonyms 26, 191
variables
integer values 192
lengths 197
pseudonyms 26
types 197
viewing conversation characteristics 19

W

Wait_For_Conversation (CMWAIT)
call description 187
example flow using 62

Communicating Your Comments to IBM

Common Programming Interface
Communications Specification

Publication No. SC31-6180-00

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM. Whichever method you choose, make sure you send your name, address, and telephone number if you would like a reply.

Feel free to comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. However, the comments you send should pertain to only the information in this manual and the way in which the information is presented. 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.

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.

If you are mailing a readers' comment form (RCF) from a country other than the United States, you can give the RCF to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:

  United States and Canada: **1-800-227-5088**

- If you prefer to send comments electronically, use this network ID:
  - IBM Mail Exchange: **USIB2HPD at IBMMAIL**
  - IBMLink: **CIBMORCF at RALVM13**
  - Internet: **USIB2HPD@VNET.IBM.COM**

Make sure to include the following in your note:

- Title and publication number of this book
- Page number or topic to which your comment applies.

Help us help you!

Common Programming Interface
Communications Specification

Publication No. SC31-6180-00

We hope you find this publication useful, readable and technically accurate, but only you can tell us! Your comments and suggestions will help us improve our technical publications. Please take a few minutes to let us know what you think by completing this form.

Overall, how satisfied are you with the information in this book?	Satisfied	Dissatisfied
	☐	☐

How satisfied are you that the information in this book is:	Satisfied	Dissatisfied
Accurate	☐	☐
Complete	☐	☐
Easy to find	☐	☐
Easy to understand	☐	☐
Well organized	☐	☐
Applicable to your task	☐	☐

Specific Comments or Problems:

Please tell us how we can improve this book:

Thank you for your response. When you send information to IBM, you grant IBM the right to use or distribute the information without incurring any obligation to you. You of course retain the right to use the information in any way you choose.

Name

Address

Company or Organization

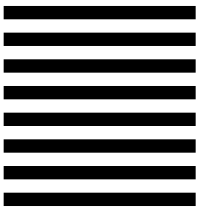
Phone No.

Help us help you!
SC31-6180-00

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

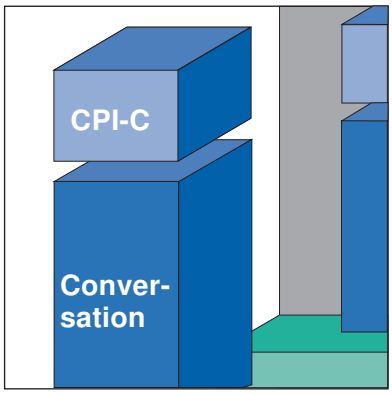
International Business Machines Corporation
Information Development
Department E15
PO BOX 12195
RESEARCH TRIANGLE PARK, NORTH CAROLINA 27709-9990

Fold and Tape **Please do not staple** Fold and Tape

Cut or Fold
Along Line

IBM®

Printed in U.S.A.



SC31-6180-00