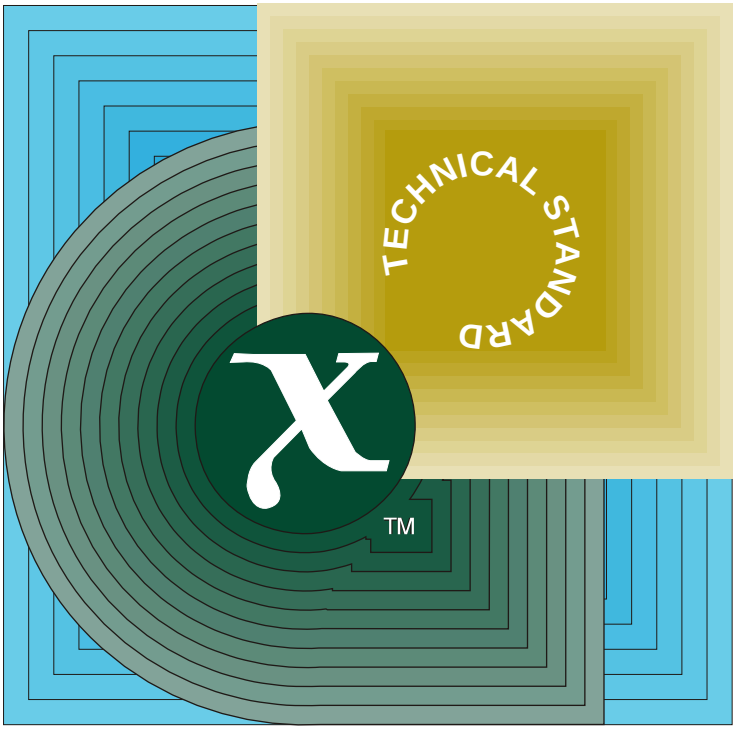
Technical Standard

Distributed Transaction Processing: The XCPI-C Specification Version 2



THE Open GROUP

[This page intentionally left blank]

X/Open CAE Specification

Distributed Transaction Processing:

CPI-C Specification, Version 2

X/Open Company Ltd.

X/Open CAE Specification

Distributed Transaction Processing: CPI-C Specification, Version 2

ISBN: 1-85912-135-7

X/Open Document Number: C419

Published by X/Open Company Ltd., U.K.

Any comments relating to the material contained in this document may be submitted to X/Open at:

X/Open Company Limited
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

XoSpecs@xopen.org

Contents

List of Figures

List of Tables

Preface

X/Open

X/Open is an independent, worldwide, open systems organisation supported by most of the world's largest information systems suppliers, user organisations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

X/Open's strategy for achieving this goal is to combine existing and emerging standards into a comprehensive, integrated, high-value and usable open system environment, called the Common Applications Environment (CAE). This environment covers the standards, above the hardware level, that are needed to support open systems. It provides for portability and interoperability of applications, and so protects investment in existing software while enabling additions and enhancements. It also allows users to move between systems with a minimum of retraining.

X/Open defines this CAE in a set of specifications which include an evolving portfolio of application programming interfaces (APIs) which significantly enhance portability of application programs at the source code level, along with definitions of and references to protocols and protocol profiles which significantly enhance the interoperability of applications and systems.

The X/Open CAE is implemented in real products and recognised by a distinctive trade mark — the X/Open brand — that is licensed by X/Open and may be used on products which have demonstrated their conformance.

X/Open Technical Publications

X/Open publishes a wide range of technical literature, the main part of which is focussed on specification development, but which also includes Guides, Snapshots, Technical Studies, Branding/Testing documents, industry surveys, and business titles.

There are two types of X/Open specification:

- *CAE Specifications*

  CAE (Common Applications Environment) specifications are the stable specifications that form the basis for X/Open-branded products. These specifications are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement an X/Open CAE specification can enjoy the benefits of a single, widely supported standard. In addition, they can demonstrate compliance with the majority of X/Open CAE specifications once these specifications are referenced in an X/Open component or profile definition and included in the X/Open branding programme.

  CAE specifications are published as soon as they are developed, not published to coincide with the launch of a particular X/Open brand. By making its specifications available in this way, X/Open makes it possible for conformant products to be developed as soon as is practicable, so enhancing the value of the X/Open brand as a procurement aid to users.

- *Preliminary Specifications*

  These specifications, which often address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations, are released in a controlled manner for the purpose of validation through implementation of products. A Preliminary specification is not a draft specification. In fact, it is as stable as X/Open can make it, and on publication has gone through the same rigorous X/Open development and review procedures as a CAE specification.

  Preliminary specifications are analogous to the *trial-use* standards issued by formal standards organisations, and product development teams are encouraged to develop products on the basis of them. However, because of the nature of the technology that a Preliminary specification is addressing, it may be untried in multiple independent implementations, and may therefore change before being published as a CAE specification. There is always the intent to progress to a corresponding CAE specification, but the ability to do so depends on consensus among X/Open members. In all cases, any resulting CAE specification is made as upwards-compatible as possible. However, complete upwards-compatibility from the Preliminary to the CAE specification cannot be guaranteed.

In addition, X/Open publishes:

- *Guides*

  These provide information that X/Open believes is useful in the evaluation, procurement, development or management of open systems, particularly those that are X/Open-compliant. X/Open Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming X/Open conformance.

- *Technical Studies*

  X/Open Technical Studies present results of analyses performed by X/Open on subjects of interest in areas relevant to X/Open's Technical Programme. They are intended to communicate the findings to the outside world and, where appropriate, stimulate discussion and actions by other bodies and the industry in general.

- *Snapshots*

  These provide a mechanism for X/Open to disseminate information on its current direction and thinking, in advance of possible development of a Specification, Guide or Technical Study. The intention is to stimulate industry debate and prototyping, and solicit feedback. A Snapshot represents the interim results of an X/Open technical activity. Although at the time of its publication, there may be an intention to progress the activity towards publication of a Specification, Guide or Technical Study, X/Open is a consensus organisation, and makes no commitment regarding future development and further publication. Similarly, a Snapshot does not represent any commitment by X/Open members to develop any specific products.

Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision, in this case as the subject technology develops and to align with emerging associated international standards. X/Open makes a distinction between revised specifications which are fully backward compatible and those which are not:

- a new *Version* indicates that this publication includes all the same (unchanged) definitive information from the previous publication of that title, but also includes extensions or additional information. As such, it *replaces* the previous publication.

- a new *Issue* does include changes to the definitive information contained in the previous publication of that title (and may also include extensions or additional information). As such, X/Open maintains *both* the previous and new issue as current publications.

Corrigenda

Most X/Open publications deal with technology at the leading edge of open systems development. Feedback from implementation experience gained from using these publications occasionally uncovers errors or inconsistencies. Significant errors or recommended solutions to reported problems are communicated by means of Corrigenda.

The reader of this document is advised to check periodically if any Corrigenda apply to this publication. This may be done either by email to the X/Open info-server or by checking the Corrigenda list in the latest X/Open Publications Price List.

To request Corrigenda information by email, send a message to info-server@xopen.co.uk with the following in the Subject line:

```
request corrigenda; topic index
```

This will return the index of publications for which Corrigenda exist.

This Document

This document is a CAE Specification (see above). It defines the Common Programming Interface for Communications (CPI-C), which is an application programming interface to a Communication Resource Manager (CRM).

This specification is structured as follows:

- Chapter 1 gives a brief overview and history of the Common Programming Interface (CPI) for Communications.
- Chapter 2 discusses the CPI Communications interface in general terms and shows its relationship to the X/Open Distributed Transaction Processing (DTP) model.
- Chapter 3 provides an overview of the CPI Communications interface and describes the basic terms and concepts used in this specification.
- Chapter 4 provides a number of sample flows that show how a program can use CPI Communications calls for program-to-program communication.
- Chapter 5 describes the format and function of each of the CPI Communications calls.
- Appendix A describes the CPI Communications variables and conversation characteristics.
- Appendix B describes the return codes that may be returned when CPI Communications calls are executed.
- Appendix C explains when the CPI Communications calls can be issued.
- Appendix D provides a mapping of CPI Communications to the services provided by the OSI TP and LU 6.2 communication resource managers.
- Appendix E contains sample CPI Communications pseudonym files for the C and COBOL programming languages.
- Appendix F contains two sample COBOL programs using CPI Communications.
- Appendix G describes application migration from the former X/Open CPI-C (X/Open CAE Specification: CPI-C 1992) to X/Open CPI-C, Version 2 (this document).

There is a glossary and an index at the end.

This specification contains both tutorial and reference information. Use the following path to achieve the most benefit:

- Read Chapter 2 for an overview of the terms and concepts used in CPI Communications. It is required to understand the sample program flows shown in Chapter 4.
- Read Chapter 4 for an explanation and examples of how to use the CPI Communications calls. When reading this chapter, use Chapter 5 to obtain additional information about the function of and required parameters for the CPI Communications calls.
- Use Chapter 5 and the appendixes for specific functional information on how to code applications.

Intended Audience

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

Although this document is an architecture specification, Chapter 4 provides a tutorial on designing application programs using CPI Communications concepts and calls.

Typographical Conventions

The following typographical conventions are used throughout this specification:

- *Italic* strings are used for emphasis and to identify the first instance of a word or phrase requiring definition.
- *Italic* strings are also used for C-language functions, for example *tx_begin*( ).
- Syntax and code examples are shown in `fixed width` font.
- Variables within syntax statements are shown in *`italic fixed width`* font.
- Any phrase that contains an underscore {_} is a pseudonym. To enhance readability, pseudonymns are used throughout this specification in place of actual call names, characteristics, variables, states, and characteristic values.
- Normal font is used for actual call names, for example CMACCP, and their pseudonyms, for example Accept_Conversation.
- *Italic* strings are used for characteristics and variables, for example *conversation_type*.
- Normal font is used for the values of characteristics and variables, for example CM_OK.
- **Bold** font is used for states, for example the **Reset** state. Bold font is also used to denote the **Backout-Required** condition.

See Section 1.3 on page 3 for further details of the typographical conventions used in this specification.

In contrast with normal X/Open style, this document uses American English spelling for consistency with the software.

Superseded Documents

Note: This specification supersedes the X/Open **Peer-to-Peer** Snapshot published in 1992.

Trade Marks

The following terms are trade marks or service marks of the IBM Corporation in the United States and other countries:

IBM
SAA
Systems Application Architecture.

Microsoft® is a registered trade mark and Windows™ is a trade mark of Microsoft Corporation.

X/Open® is a registered trade mark, and the ''X'' device is a trade mark, of X/Open Company Limited.

Acknowledgements

X/Open gratefully acknowledges the collaboration with and contributions from the CPI-C Implementers' Workshop (CIW).

Referenced Documents

The following standards are referenced in this specification:

ISO 7498

ISO 7498, Information Processing Systems — Open Systems Interconnection, Parts 1 and 3:

Part 1: 1994, The Basic Model
Part 3: 1989, Naming and Addressing.

ISO/IEC 9545

ISO/IEC 9545: 1989, Information Technology — Open Systems Interconnection — Application Layer Structure.

ISO/IEC 9594

ISO/IEC 9594: 1990, Information Technology — Open Systems Interconnection — The Directory, Parts 1 to 8:

Part 1: Overview of Concepts, Models and Services (CCITT X.500)
Part 2: Models (CCITT X.501)
Part 3: Abstract Service Definition (CCITT X.511)
Part 4: Procedures for Distributed Operation (CCITT X.518)
Part 5: Protocol Specifications (CCITT X.519)
Part 6: Selected Attribute Types (CCITT X.520)
Part 7: Selected Object Classes (CCITT X.521)
Part 8: Authentication Framework (CCITT X.509).

OSI TP

ISO/IEC 10026, Information Technology — Open Systems Interconnection — Distributed Transaction Processing, Parts 1 to 6:

Part 1: 1992, OSI TP Model
Part 2: 1992, OSI TP Service
Part 3: 1992, Protocol Specification
Part 4: 1995, Protocol Implementation Conformance Statement (PICS) proforma
Part 5: DIS 1993, Application context proforma and guidelines when using OSI TP
Part 6: 1994, Unstructured Data Transfer.

OSI TP Profiles

ISO/IEC ISP 12061: 1995, Information Technology — Open Systems Interconnection — International Standardized Profiles: OSI Distributed Transaction Processing, Parts 1 to 10:

Part 1: Introduction to the Transaction Processing Profiles
Part 2: Support of OSI TP APDUs
Part 3: Support of the CCR APDUs
Part 4: Support of Session, Presentation and ACSE PDUs
Part 5: Application supported transactions — Polarized control (ATP11)
Part 6: Application supported transactions — Shared control (ATP12)
Part 7: Provider supported unchained transactions — Polarized control (ATP21)
Part 8: Provider supported unchained transactions — Shared control (ATP22)
Part 9: Provider supported chained transactions — Polarized control (ATP31)
Part 10: Provider supported chained transactions — Shared control (ATP32).

T.61
: CCITT Recommendation T.61: 1984, Character Repertoire and Coded Character Sets for the International Teletex Service.

The following IBM and CPI-C Implementers' Workshop (CIW) specifications are referenced in this specification:

APPC
: Advanced Peer to Peer Communications: Resource Reference, Order Number G325-0055), IBM Corporation.

CPI-C 1.0
: Systems Application Architecture Common Programming Interface Communications Reference, First Edition (May 1988), Order Number SC26-4399-00, IBM Corporation.

CPI-C 1.1
: Systems Application Architecture Common Programming Interface Communications Reference, Third Edition (August 1990), Order Number SC26-4399-02, IBM Corporation.

CPI-C 1.2
: Common Programming Interface Communications Specification, First Edition (March 1993), Order Number SC31-6180-00, IBM Corporation.

CPI-C 2.0
: Common Program Interface Communications Specification, Second Edition (June 1994), Order Number SC31-6180-01, IBM Corporation.

SAA
: System Application Architecture: Common Programming Interface: Communications Reference (Seventh Edition), Order Number SC26-4399-06, IBM Corporation.

  System Application Architecture: Common Programming Interface: Resource Recovery Reference, Order Number (SC31-6821), IBM Corporation.

SNA
: System Network Architecture: LU 6.2 Reference: Peer Protocols, Order Number SC31-6808, IBM Corporation.

  System Network Architecture: Transaction Programmer's Reference Manual for LU Type 6.2, Order Number GC30-3084, IBM Corporation.

  System Network Architecture: Formats manual, Order Number GA27-3136, IBM Corporation.

The following X/Open documents are referenced in this specification:

CPI-C
: X/Open CAE Specification, February 1992, CPI-C (ISBN: 1-872630-35-9, C210).

DTP
: X/Open Guide, November 1993, Distributed Transaction Processing: Reference Model, Version 2 (ISBN: 1-85912-019-9, G307).

Peer-to-Peer
: X/Open Snapshot, December 1992, Distributed Transaction Processing: The Peer-to-Peer Specification (ISBN: 1-872630-79-0, S214).

TX

X/Open CAE Specification, April 1995, Distributed Transaction Processing: The TX (Transaction Demarcation) Specification (ISBN: 1-85912-094-6, C504).

TxRPC

X/Open CAE Specification, October 1995, Distributed Transaction Processing: The TxRPC Specification (ISBN: 1-85912-115-2, C505).

XA

X/Open CAE Specification, December 1991, Distributed Transaction Processing: The XA Specification (ISBN: 1-872630-24-3, C193 or XO/CAE/91/300).

XA+

X/Open Snapshot, July 1994, Distributed Transaction Processing: The XA+ Specification, Version 2 (ISBN: 1-85912-046-6, S423).

XAP-TP

X/Open CAE Specification, April 1995, ACSE/Presentation: Transaction Processing API (XAP-TP) (ISBN: 1-85912-091-1, C409).

XATMI

X/Open CAE Specification, October 1995, Distributed Transaction Processing: The XATMI Specification (ISBN: 1-85912-130-6, C506).

XDCS

X/Open Guide, November 1992, Distributed Computing Services (XDCS) Framework (ISBN: 1-872630-64-2, G212).

Chapter 1

Introduction

This chapter provides an outline of the X/Open Distributed Transaction Processing Model and explains the position of this specification as one of the Communication Resource Manager (CRM) interfaces. This chapter also gives a brief history of the development of the CPI-C interface, and provides guidance on the stylistic conventions used to represent call names, call pseudonyms, characteristics, variables, values, states and queues.

1.1 X/Open DTP Model

The X/Open Distributed Transaction Processing (DTP) model is a software architecture that allows multiple application programs to share resources provided by multiple resource managers, and allows their work to be coordinated into global transactions.

The X/Open DTP model comprises five basic functional components:

- an Application Program (AP), which defines transaction boundaries and specifies actions that constitute a transaction
- Resource Managers (RMs) such as databases or file access systems, which provide access to resources
- a Transaction Manager (TM), which assigns identifiers to transactions, monitors their progress, and takes responsibility for transaction completion and for coordinating failure recovery
- Communication Resource Managers (CRMs), which control communication between distributed applications within or across TM domains
- a communication protocol, which provides the underlying communication services used by distributed applications and supported by CRMs.

X/Open DTP publications based on this model specify portable Application Programming Interfaces (APIs) and system-level interfaces that facilitate:

- portability of application program source code to any X/Open environment that offers those APIs
- interchangeability of TMs, RMs and CRMs from various sources
- interoperability of diverse TMs, RMs and CRMs in the same global transaction.

Chapter 2 defines each component in more detail and illustrates the flow of control.

1.2 X/Open Communication Resource Manager Interfaces

An important aspect of distributed transaction processing applications is communication. Within the product domain for DTP tools, there are several popular communication paradigms in common use today or expected to be in common use in the future. The communication paradigm chosen can significantly influence the architecture of the application. The unique strengths of each paradigm make it attractive for specific applications.

The referenced **DTP** guide defines a functional component known as a Communication Resource Manager (CRM), which provides access to a communication medium between applications.

Because it is not possible to choose a single communication paradigm applicable to the entire broad range of DTP applications, X/Open provides application programming interfaces (APIs) for the most popular paradigms in order to bring the benefits of open systems to the widest possible range of transaction processing applications. These are the request/response paradigm and the conversational paradigm.

Many applications already running on open systems use the request/response paradigm. X/Open specifications for this paradigm are the library-based XATMI CRM interface (see the referenced **XATMI** specification) and the IDL-based TxRPC CRM interface (see the referenced **TxRPC** specification). TxRPC fits within the context of the X/Open Distributed Computing Services Framework (XDCS) and allows application writers to invoke remote procedure calls (RPCs) in the same form as local procedures, but with transaction semantics.

For applications choosing to use the conversational paradigm, where communication takes place through an application-defined exchange of messages, X/Open offers the library-based interfaces XATMI (see the referenced **XATMI** specification) and CPI-C (see this document).

The conversational model of program-to-program communication 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 CPI-C conversational model is implemented in two major communication protocols: Open Systems Interconnection Distributed Transaction Processing (OSI TP)[1] and Advanced Program-to-Program Communications (APPC). The APPC model is also referred to as logical unit type 6.2 (LU 6.2). CPI-C Version 2 provides access to both communication 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 shields the program from details of system connectivity and eases the integration and porting of the application programs across the supported environments.

1. See the referenced OSI TP standards.

1.3 Naming Conventions: Calls, Characteristics, Variables and Values

Pseudonyms for the actual calls, characteristics, variables, states, and characteristic values that make up CPI Communications are used throughout this specification to enhance understanding and readability. Where possible, underscores (_) and complete names are used in the pseudonyms. Any phrase in the specification 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 specification uses the following conventions to aid in distinguishing between the four types of pseudonyms:

- *Calls* are shown in all capital letters. Each underscore-separated portion of a call's pseudonym begins with a capital letter. 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 upper-case letters (such as CM_OK) and represent actual integer values that are to be placed into the variable. For a list of the integer values that are placed in the variables, see Table A-1 on page 330.
- *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. Bold is also used to denote the **Backout-Required** condition.
- *Queues* are used to group related CPI Communications calls. Queue names begin with capital letters. The parts of a queue name are connected with a hyphen.

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

- Chapter 5 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 that CMSRC has a parameter list of *conversation_ID*, *return_control*, and *return_code*.
- Appendix A provides a complete description of all variables used in the specification 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 A-3 on page 341.
- Table A-1 on page 330 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. 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. Section 5.2 on page 117 provides information on system files that can be used to establish pseudonyms for a program.

1.4 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 communication programming. To date, there have been five versions of the interface:

- CPI Communications as it was first introduced in 1987, referred to in this specification as CPI-C 1.0
- extensions made to CPI-C 1.0 by IBM, CPI-C 1.1
- extensions made by X/Open (see the referenced **CPI-C CAE** specification)
- CPI Communications extensions to support X/Open, CPI-C 1.2
- CPI Communications extensions made by the CPI-C Implementers' Workshop (CIW), CPI-C 2.0.

This specification documents X/Open CPI-C Version 2, which derives from the above CPI-C 2.0 but with the following major differences:

- X/Open CPI-C, Version 2 only supports the C and COBOL programming languages.
- X/Open CPI-C, Version 2 does not support the concept of a *distributed directory*.

Note: This specification supersedes the X/Open **Peer-to-Peer Snapshot** published in 1992.

1.5 Functional Levels of CPI Communications

The following sections list the major features of each level of the CPI Communications architecture.

1.5.1 CPI-C 1.0

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

- 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).

1.5.2 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 referenced SAA CPI-C specification.

1.5.3 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 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.

For further details, see the referenced **CPI-C CAE** specification.

1.5.4 CPI-C 1.2

CPI-C 1.0 was designed to provide a consistent programming interface for communication 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 consolidated 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.

1.5.5 CPI-C 2.0

CPI-C 2.0 provides enhancements to some CPI-C 1.2 functions as well as offering several new functions:

- support for full-duplex conversations and expedited data
- enhanced support for non-blocking processing with the addition of queue-level processing and a callback function
- support for OSI TP applications
- support for secondary information to determine the cause of a return code.

1.5.6 Call Table

Table 1-1 lists the calls defined for the different versions of CPI Communications. For CPI-C versions 1.0, 1.1, 1.2 and 2.0, an X indicates that the call was part of that specific version. The column marked **CPI-C X/Open** refers to the original version of X/Open CPI-C. In this column, the actual names of the calls from that version are given in place of each X.

Table 1-1 Versions of CPI Communications

Call Name	CPI-C 1.0	CPI-C 1.1	CPI-C X/Open	CPI-C 1.2	CPI-C 2.0
Starter Set					
Accept_Conversation	X	X	CMACCP	X	X
Allocate	X	X	CMALLC	X	X
Deallocate	X	X	CMDEAL	X	X
Initialize_Conversation	X	X	CMINIT	X	X
Receive	X	X	CMRCV	X	X
Send_Data	X	X	CMSEND	X	X
Advanced Function					
for synchronization and control:					
Confirm	X	X	CMCFM	X	X
Confirmed	X	X	CMCFMD	X	X
Deferred_Deallocate					X
Flush	X	X	CMFLUS	X	X
Include_Partner_In_Transaction					X
Prepare					X
Prepare_To_Receive	X	X	CMPTR	X	X
Receive_Expedited_Data					X
Request_To_Send	X	X	CMRTS	X	X
Send_Error	X	X	CMSERR	X	X
Send_Expedited_Data					X
Test_Request_To_Send_Received	X	X	CMTRTS	X	X

Call Name	CPI-C 1.0	CPI-C 1.1	CPI-C X/Open	CPI-C 1.2	CPI-C 2.0
for modifying conversation characteristics:					
Set_AE_Qualifier					X
Set_Allocate_Confirm					X
Set_AP_Title					X
Set_Application_Context_Name					X
Set_Begin_Transaction					X
Set_Confirmation_Urgency					X
Set_Conversation_Security_Password			CMSCSP	X	X
Set_Conversation_Security_Type			CMSCST	X	X
Set_Conversation_Security_User_ID			CMSCSU	X	X
Set_Conversation_Type	X	X	CMSCT	X	X
Set_Deallocate_Type	X	X	CMSDT	X	X
Set_Error_Direction	X	X	CMSED	X	X
Set_Fill	X	X	CMSF	X	X
Set_Initialization_Data					X
Set_Join_Transaction					X
Set_Log_Data	X	X	CMSLD	X	X
Set_Mode_Name	X	X	CMSMN	X	X
Set_Partner_LU_Name	X	X	CMSPLN	X	X
Set_Prepare_Data_Permitted					X
Set_Prepare_To_Receive_Type	X	X	CMSPTR	X	X
Set_Receive_Type	X	X	CMSRT	X	X
Set_Return_Control	X	X	CMSRC	X	X
Set_Send_Receive_Mode					X
Set_Send_Type	X	X	CMSST	X	X
Set_Sync_Level					
CM_NONE	X	X	CMSSL	X	X
CM_CONFIRM	X	X	CMSSL	X	X
CM_SYNC_POINT		X		X	X
CM_SYNC_POINT_NO_CONFIRM					X
Set_TP_Name	X	X	CMSTPN	X	X
Set_Transaction_Control					X

Call Name	CPI-C 1.0	CPI-C 1.1	CPI-C X/Open	CPI-C 1.2	CPI-C 2.0
for examining information about the conversation and CRM:					
Extract_AE_Qualifier					X
Extract_AP_Title					X
Extract_Application_Context_Name					X
Extract_Conversation_State		X		X	X
Extract_Conversation_Type	X	X	CMECT	X	X
Extract_Initialization_Data					X
Extract_Maximum_Buffer_Size				X	X
Extract_Mode_Name	X	X	CMEMN	X	X
Extract_Partner_LU_Name	X	X	CMEPLN	X	X
Extract_Secondary_Information					X
Extract_Security_User_ID				X	X
Extract_Send_Receive_Mode					X
Extract_Sync_Level	X	X	CMESL	X	X
Extract_TP_Name			CMETPN	X	X
Extract_Transaction_Control					X
for non-blocking operations:					
Cancel_Conversation			CMCANC	X	X
Set_Processing_Mode			CMSPM	X	X
Set_Queue_Callback_Function					X
Set_Queue_Processing_Mode					X
Wait_For_Completion					X
Wait_For_Conversation			CMWAIT	X	X
for accepting multiple conversations:					
Accept_Incoming				X	X
Initialize_For_Incoming				X	X
Release_Local_TP_Name				X	X
Specify_Local_TP_Name			CMSLTP	X	X
for data conversion:					
Convert_Incoming			CMCNVI	X	X
Convert_Outgoing			CMCNVO	X	X

Chapter 2

Model and Definitions

This chapter discusses the CPI-C interface in general terms and provides necessary background material for the rest of the specification. The chapter shows the relationship of the interface to the X/Open DTP model. The chapter also states the design assumptions that the interface uses and shows how the interface addresses common DTP concepts.

2.1 X/Open DTP Model

The boxes in the figure below are the functional components and the connecting lines are the interfaces between them. The arrows indicate the directions in which control may flow.

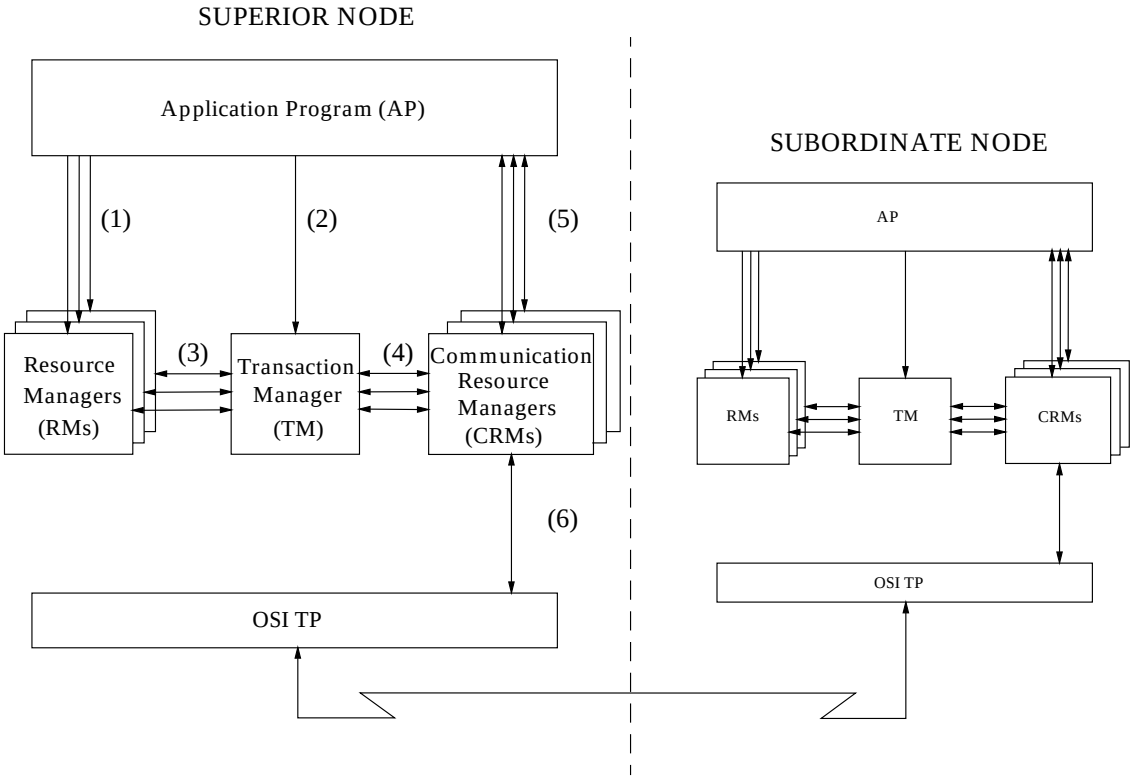


Figure 2-1 Functional Components and Interfaces

Descriptions of the functional components shown can be found in Section 2.1.1 on page 12. The numbers in brackets in the above figure represent the different X/Open interfaces that are used in the model. They are described in Section 2.1.2 on page 13.

For more details of this model and diagram, including detailed definitions of each component, see the referenced **DTP** guide.

2.1.1 Functional Components

Application Program (AP)

The application program (AP) implements the desired function of the end-user enterprise. Each AP specifies a sequence of operations that involves resources such as databases. An AP defines the start and end of global transactions, accesses resources within transaction boundaries, and normally makes the decision whether to commit or roll back each transaction.

Where two or more APs cooperate within a global transaction, the X/Open DTP model supports three *paradigms* for AP to AP communication. These are the TxRPC, XATMI and CPI-C interfaces.

Transaction Manager (TM)

The transaction manager (TM) manages global transactions and coordinates the decision to start them, and commit them or roll them back. This ensures atomic transaction completion. The TM also coordinates recovery activities of the resource managers when necessary, such as after a component fails.

Resource Manager (RM)

The resource manager (RM) manages a defined part of the computer's shared resources. These may be accessed using services that the RM provides. Examples for RMs are database management systems (DBMSs), a file access method such as X/Open ISAM, and a print server.

In the X/Open DTP model, RMs structure all changes to the resources they manage as recoverable and atomic transactions. They let the TM coordinate completion of these transactions atomically with work done by other RMs.

Communication Resource Manager (CRM)

A CRM allows an instance of the model to access another instance either inside or outside the current TM Domain. Within the X/Open DTP model, CRMs use OSI TP services to provide a communication layer across TM Domains. CRMs aid global transactions by supporting the following interfaces:

- the communication paradigm (TxRPC, XATMI or CPI-C) used between an AP and CRM
- XA+ communication between a TM and CRM
- XAP-TP communication between a CRM and OSI TP.

A CRM may support more than one type of communication paradigm, or a TM Domain may use different CRMs to support different paradigms. The XA+ interface provides global transaction information across different instances and TM Domains. The CRM allows a global transaction to extend to another TM Domain, and allows TMs to coordinate global transaction commit and abort requests from (usually) the superior AP. Using the above interfaces, information flows from superior to subordinate and *vice versa*.

2.1.2 Interfaces between Functional Components

There are six interfaces between software components in the X/Open DTP model. The numbers correspond to the numbers in Figure 2-1 on page 11.

(1) **AP-RM.** The AP-RM interfaces give the AP access to resources. X/Open interfaces, such as SQL and ISAM, provide AP portability. The X/Open DTP model imposes few constraints on native RM APIs. The constraints involve only those native RM interfaces that define transactions. (See the referenced **XA** specification.)

(2) **AP-TM.** The AP-TM interface (the TX interface) provides the AP with an Application Programming Interface (API) by which the AP coordinates global transaction management with the TM. For example, when the AP calls *tx_begin*( ) the TM informs the participating RMs of the start of a global transaction. After each request is completed, the TM provides a return value to the AP reporting back the success or otherwise of the TX call.

For details of the AP-TM interface, see the referenced **TX** (Transaction Demarcation) specification.

(3) **TM-RM.** The TM-RM interface (the XA interface) lets the TM structure the work of RMs into global transactions and coordinate completion or recovery. The XA interface is the bidirectional interface between the TM and RM.

The functions that each RM provides for the TM are called the *xa_**( ) functions. For example the TM calls *xa_start*( ) in each participating RM to start an RM-internal transaction as part of a new global transaction. Later, the TM may call in sequence *xa_end*( ) *xa_prepare*( ) and *xa_commit*( ) to coordinate a (successful in this case) two-phase commit protocol. The functions that the TM provides for each RM are called the *ax_**( ) functions. For example an RM calls *ax_reg*( ) to register dynamically with the TM.

For details of the TM-RM interface, see the referenced **XA** specification.

(4) **TM-CRM.** The TM-CRM interface (the XA+ interface) supports global transaction information flow across TM Domains. In particular TMs can instruct CRMs by use of *xa_**( ) function calls to suspend or complete transaction branches, and to propagate global transaction commitment protocols to other transaction branches. CRMs pass information to TMs in subordinate branches by use of *ax_**( ) function calls. CRMs also use *ax_**( ) function calls to request the TM to create subordinate transaction branches, to save and retrieve recovery information, and to inform the TM of the start and end of blocking conditions.

For details of the TM-CRM interface, see the referenced **XA+** specification.

The XA+ interface is a superset of the XA interface and supersedes its purpose. Since the XA+ interface is invisible to the AP, the TM and CRM may use other methods to interconnect without affecting application portability.

(5) **AP-CRM.** X/Open provides portable APIs for DTP communication between APs within a global transaction. The API chosen can significantly influence (and may indeed be fundamental to) the whole architecture of the application. For this reason, these APIs are frequently referred to in this specification and elsewhere as *communication paradigms*. In practice, each paradigm has unique strengths, so X/Open offers the following popular paradigms:

— the TxRPC interface (see the **TxRPC** specification)

— the XATMI interface (see the **XATMI** specification)

— the CPI-C interface (see this document).

X/Open interfaces, such as the three CRM APIs listed above, provide application portability across products offering the same CRM API. The X/Open DTP model imposes few constraints on native CRM APIs.

(6) **CRM-OSI TP.** This interface (the XAP-TP interface) provides a programming interface between a CRM and Open Systems Interconnection Distributed Transaction Processing (OSI TP) services. XAP-TP interfaces with the OSI TP Service and the Presentation Layer of the seven-layer OSI model. X/Open has defined this interface to support portable implementations of application-specific OSI services. The use of OSI TP is mandatory for communication between heterogeneous TM domains. For details of this interface, see the referenced **XAP-TP** specification and OSI TP standards.

2.2 Definitions

For additional definitions see the referenced **DTP** guide.

2.2.1 Transaction

A transaction is a complete unit of work. It may comprise many computational tasks, which may include user interface, data retrieval, and communication. A typical transaction modifies shared resources. (The OSI TP standards (model) define transactions more precisely.)

Transactions must be able to be *rolled back*. A human user may roll back the transaction in response to a real-world event, such as a customer decision. A program can elect to roll back a transaction. For example, account number verification may fail or the account may fail a test of its balance. Transactions also roll back if a component of the system fails, keeping it from retrieving, communicating, or storing data. Every DTP software component subject to transaction control must be able to undo its work in a transaction that is rolled back at any time.

When the system determines that a transaction can complete without failure of any kind, it *commits* the transaction. This means that changes to shared resources take permanent effect. Either commitment or rollback results in a consistent state. *Completion* means either commitment or rollback.

2.2.2 Transaction Properties

Transactions typically exhibit the following properties:

Atomicity This means that the results of the transaction's execution are either all committed or all rolled back.

Consistency This means that a completed transaction transforms a shared resource from one valid state to another valid state.

Isolation This means that changes to shared resources that a transaction effects do not become visible outside the transaction until the transaction commits.

Durability This means the changes that result from transaction commitment survive subsequent system or media failures.

These properties are known by their initials as the **ACID** properties. In the X/Open DTP model, the TM coordinates Atomicity at global level whilst each RM is responsible for the Atomicity, Consistency, Isolation and Durability of its resources.

2.2.3 Distributed Transaction Processing

Within the scope of this document, DTP systems are those where work in support of a single transaction may occur across RMs. This has several implications:

- The system must have a way to refer to a transaction that encompasses all work done anywhere in the system.
- The decision to commit or roll back a transaction must consider the status of work done anywhere on behalf of the transaction. The decision must have uniform effect throughout the DTP system.

Even though an RM may have an X/Open-compliant interface such as Structured Query Language (SQL), it must also address these two items to be useful in the DTP environment.

2.2.4 Global Transactions

Every RM in the DTP environment must support transactions as described in Section 2.2.1 on page 15. Many RMs already structure their work into recoverable units.

In the DTP environment, many RMs may operate in support of the same unit of work. This unit of work is a *global transaction*. For example, an AP might request updates to several different databases. Work occurring anywhere in the system must be committed atomically. Each RM must let the TM coordinate the RM's recoverable units of work that are part of a global transaction.

Commitment of an RM's internal work depends not only on whether its own operations can succeed, but also on operations occurring at other RMs, perhaps remotely. If any operation fails anywhere, every participating RM must roll back all operations it did on behalf of the global transaction. A given RM is typically unaware of the work that other RMs are doing. A TM informs each RM of the existence, and directs the completion, of global transactions. An RM is responsible for mapping its recoverable units of work to the global transaction.

2.2.5 Transaction Branches

A global transaction has one or more *transaction branches* (or *branches*). A branch is a part of the work in support of a global transaction for which the TM and the RM engage in a separate but coordinated transaction commitment protocol. Each of the RM's internal units of work in support of a global transaction is part of exactly one branch.

A global transaction might have more than one branch when, for example, the AP uses a CRM to communicate with remote applications. The CRM asks the TM to create a new transaction branch prior to accessing a remote AP for the first time. Subsequent accesses to the same remote AP are typically done within the same transaction branch. Accesses to different remote APs are typically done in separate transaction branches.

After the TM begins the transaction commitment protocol, the RM receives no additional work to do on that transaction branch. The RM may receive additional work on behalf of the same transaction, from different branches. The different branches are related in that they must be completed atomically. However, the TM directs the commitment protocol for each branch separately. That is, an RM receives a separate commitment request for each branch.

Chapter 3

Interface Overview

This chapter gives an overview of the CPI-C interface. In an X/Open DTP system, CPI-C is the interface between an AP and a CRM.

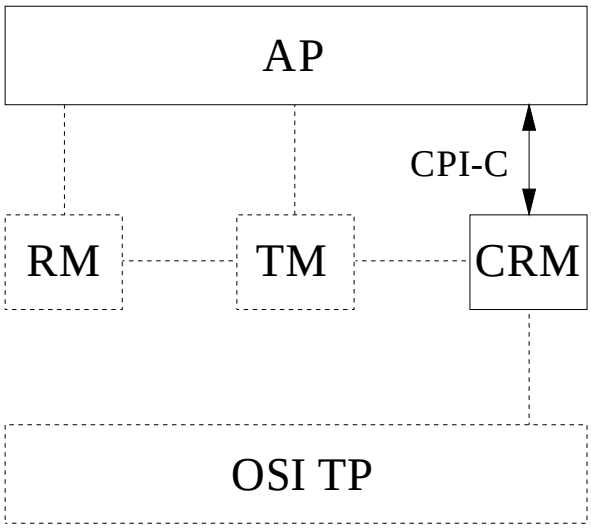


Figure 3-1 The CPI-C Interface

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 inter-program services, including:

- sending and receiving data
- synchronizing processing between programs
- notifying a partner of errors in the communication.

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

3.1 Communication across a Network

Figure 3-2 illustrates the logical view of a sample network. It consists of three *communication resource managers*[2] (CRMs): CRM X, CRM Y, and CRM Z. Each CRM has two *logical connections* with two other CRMs; the logical connections are shown as the gray portions of Figure 3-2 and enable communication between the CRMs. The network shown in Figure 3-2 is a simple one. In a real network, the number of CRMs and logical connections between the CRMs can be in the thousands.

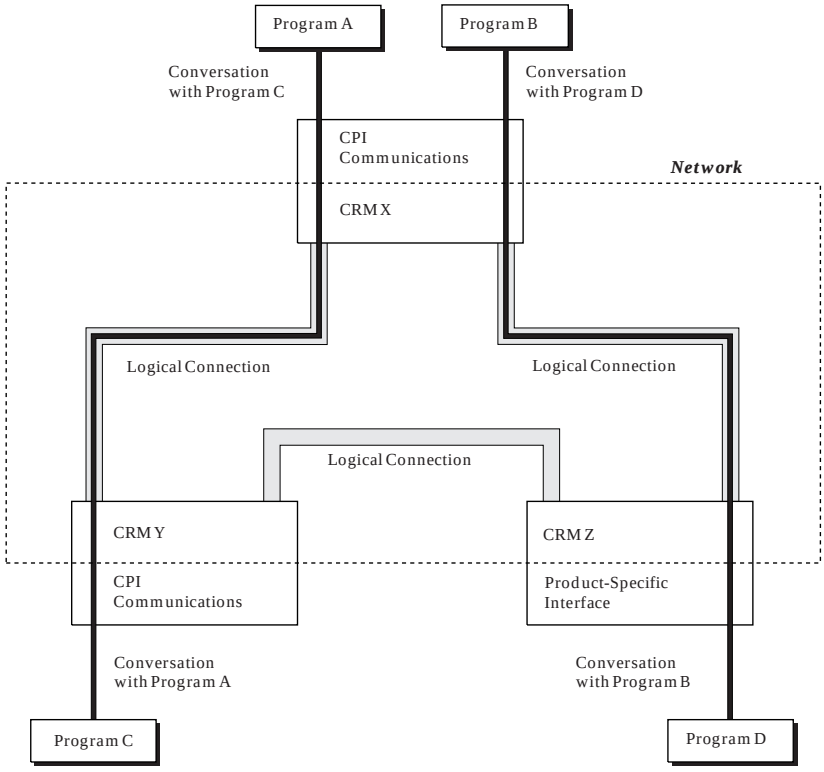


Figure 3-2 Programs Using CPI Communications to Converse through a Network

The CRMs and the logical connections shown in Figure 3-2 are generic representations of real networks. In an OSI network, CRMs are called *application entities* and the logical connections are *associations*. If this were an SNA network, the CRMs would be referred to as *logical units* of type 6.2 and the logical connections would be *sessions*. The physical network, which consists of nodes (processors) and data links between nodes, is not shown in Figure 3-2 because a program using CPI Communications does not *see* these resources. A program uses the logical network of CRMs, which in turn communicates with and uses the physical network. The CRMs discussed in this specification are of type OSI TP or type LU 6.2.

2. Communication resource managers can provide many functions in a network. In this specification, the term *communication resource manager* refers only to resource managers that provide conversation services to CPI Communications programs. Other CRMs might, for example, provide services for remote procedure calls or message-queuing interfaces.

3.2 Conversation Types

Just as two CRMs communicate using a logical connection, two programs exchange data using a *conversation*. For example, the conversation between Program A and Program C is shown in Figure 3-2 on page 18 as a single bold line between the two programs. The line indicating the conversation is shown on top of the logical connection because a conversation allows programs to communicate *over* the logical connection between the CRMs.

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 logical 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*.

  Notes:

  1. 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 **APPLICATION USAGE** section of *Send_Data (CMSEND)* on page 230.
  2. Because OSI TP CRMs do not exchange the conversation characteristic that determines whether a conversation is to be mapped or basic, the remote application must also issue a Set_Conversation_Type call when basic conversations are being used, to override the default value of CM_MAPPED_CONVERSATION for the *conversation_type* conversation characteristic.

3.3 Send-Receive Modes

CPI Communications supports two modes for sending and receiving data on a conversation:

Half-duplex
: Only one of the programs has *send control*, the right to send data, at any time. Send control must be transferred to the other program before that program can send data.

Full-duplex
: Both programs can send and receive data at the same time. Thus, both programs have send control.

The *send-receive mode* on a conversation is determined at the time the conversation is established.

For further information, see Section 3.8.2 on page 36.

3.4 Program Partners

Two programs involved in a conversation are called *partners* in the conversation. If a CRM-CRM logical connection exists, or can be made to exist, between the nodes containing the partner programs, two programs can communicate through the network with 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 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 a product-specific programming interface for a particular environment and not CPI Communications. The conversation between Program B and Program D in Figure 3-2 on page 18 is an example of such a situation. Some restrictions may apply in this situation, since CPI Communications does not support all available network functions. See Appendix D for a more complete discussion.

3.4.1 Identifying the Partner Program

CPI Communications requires a certain amount of destination information, such as the name of the partner program and the name of the CRM at the partner's node, before it can establish a conversation. Sources for this information include:

Program-supplied
: The program can supply the destination information directly.

Side information
: The program can use data contained in local *side information*. The side information is accessed using an 8-byte symbolic destination name or *sym_dest_name*, which identifies the partner program.

There are some considerations to keep in mind when using the different techniques:

- Program-supplied information may require recompilation if the address of the partner program changes.
- Use of *sym_dest_name* allows only a small name space of locally-defined names.
- Side information requires local administration on each system.
- Movement of a program may result in update of side information on multiple systems.

3.5 Operating Environment

Figure 3-3 gives a more detailed view of Program A's operating environment. As in Figure 3-2 on page 18, the bold black line shows the conversation Program A has established with its partner program. The new line between the program and CPI Communications represents Program A's use of program calls to communicate with CPI Communications. The different types of CPI Communications calls are discussed in Section 3.6 on page 24.

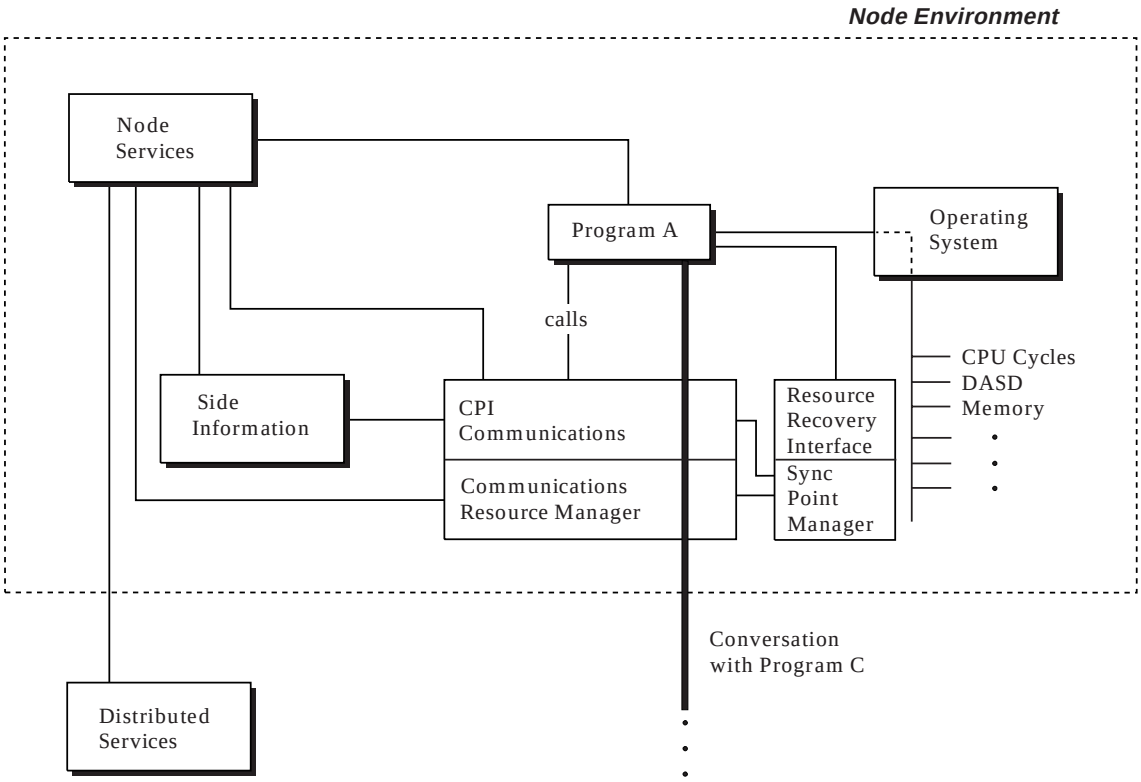


Figure 3-3 Operating Environment of CPI Communications Program

In addition to the new line with CPI Communications, Figure 3-3 also shows Program A using several other generic elements:

- node services
- side information
- distributed services
- resource recovery interface
- operating system.

These elements are discussed in the following sections.

3.5.1 Node Services

Node services represents a number of *utility* functions within the local system environment that are available for CPI Communications and other programming interfaces. 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 includes the following general functions:

- Setting and accessing of side information

  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 Section 3.8 on page 29. (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 Section 4.2 on page 64) or by local (operator) action. In either case, node services sets up the data paths and operating environment required by the program, validates and establishes security parameters under which the program executes, 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 abnormally deallocates any dangling conversations.

- Acquiring and validating access security information

  Node services provides interfaces for CRMs both to acquire and to validate access security information on behalf of a user. See Section 3.11 on page 47 for more information.

3.5.2 Side Information

As discussed in Section 3.4.1 on page 20, CPI Communications allows a program to identify its partner program with a *sym_dest_name*. The *sym_dest_name* is provided on the Initialize_Conversation call and corresponds to a side-information entry containing destination information for the partner program. The information that needs to be specified in the side-information entry depends on the type of CRM (OSI TP or LU 6.2) required to contact the program. Each piece of information may have associated attributes such as length and format for *AP_title* and *AE_qualifier*.

Here is the possible information if the CRM type is OSI TP:

- *AP_title*

  When combined with the *AE_qualifier*, the application-process-title indicates the name of the application-entity where the partner program is located. The *AP_title* combined with an *AE_qualifier* is equivalent to a fully qualified *partner_LU_name* in SNA.

- *AE_qualifier*

  Indicates the application-entity-qualifier, which is used to distinguish between application-entities having the same *AP_title*, if required.

- *application_context_name*

  Specifies the name of the application context being used on the conversation. An *application context* is a set of operating rules that two programs have agreed to follow.

Here is the possible information if the CRM type is LU 6.2:

- *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.

In addition, the entry may contain the following information, which are not CRM-type dependent.

- *TP_name*

  Specifies the name of the remote program. *TP_name* stands for *transaction program name*. In this specification, *transaction program*, *application program*, and *program* are synonymous, all denoting a program using CPI Communications. See Appendix D for details of how a CPI Communications program can interact with non-CPI Communications programs.

- *mode_name*

  Used to designate the properties of the logical connection to be established 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 CRM to establish logical connections with its partners.

- *conversation_security_type*

  Specifies the type of access security information to be included in the conversation startup request. See Section 3.11 on page 47 for more information.

- *security_user_ID*

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

- *security_password*

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

Programs not wanting to use side information can specify a *sym_dest_name* of blanks on the Initialize_Conversation call. For more information, see *Initialize_Conversation (CMINIT)* on page 195.

3.6 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 Section 3.8 on page 29.

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 an 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 *control_information_received*, *data_received*, and *status_received* parameters. Additionally, the return code may be associated with *secondary information*, which can be used to determine the cause of the return code.

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 Section 4.2 on page 64.

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 Section 4.3 on page 70.

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 3-1 on page 25 lists the two groups of CPI Communications calls.

Starter Set	
Initialize_Conversation	
Accept_Conversation	
Allocate	
Send_Data	
Receive	
Deallocate	
Advanced Function	
Accept_Incoming	Set_AE_Qualifier
Cancel_Conversation	Set_Allocate_Confirm
Confirm	Set_AP_Title
Confirmed	Set_Application_Context_Name
Convert_Incoming	Set_Begin_Transaction
Convert_Outgoing	Set_Confirmation_Urgency
Deferred_Deallocate	Set_Conversation_Security_Password
Flush	Set_Conversation_Security_Type
Include_Partner_In_Transaction	Set_Conversation_Security_User_ID
Initialize_For_Incoming	Set_Conversation_Type
Prepare	Set_Deallocate_Type
Prepare_To_Receive	Set_Error_Direction
Receive_Expedited_Data	Set_Fill
Release_Local_TP_Name	Set_Initialization_Data
Request_To_Send	Set_Join_Transaction
Send_Error	Set_Log_Data
Send_Expedited_Data	Set_Mode_Name
Specify_Local_TP_Name	Set_Partner_LU_Name
Test_Request_To_Send_Received	Set_Prepare_Data_Permitted
Wait_For_Completion	Set_Prepare_To_Receive_Type
Wait_For_Conversation	Set_Processing_Mode
	Set_Queue_Callback_Function
Extract_AE_Qualifier	Set_Queue_Processing_Mode
Extract_AP_Title	Set_Receive_Type
Extract_Application_Context_Name	Set_Return_Control
Extract_Conversation_State	Set_Send_Receive_Mode
Extract_Conversation_Type	Set_Send_Type
Extract_Initialization_Data	Set_Sync_Level
Extract_Maximum_Buffer_Size	Set_TP_Name
Extract_Mode_Name	Set_Transaction_Control
Extract_Partner_LU_Name	
Extract_Secondary_Information	
Extract_Security_User_ID	
Extract_Send_Receive_Mode	
Extract_Sync_Level	
Extract_TP_Name	
Extract_Transaction_Control	

Table 3-1 Breakdown of Calls between Starter Set and Advanced Function

A list of the calls and a brief description of each call's function is provided in Table 5-1 on page 120.

3.7 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 uses this *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 CRM. 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 uses 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 Section 3.8 on page 29.

3.7.1 Multiple Conversations

In the previous example, 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.

3.7.1.1 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 the 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.

3.7.1.2 Multiple Outbound Conversations

Figure 3-4 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 knows that these calls apply to the conversation with Program B. Similarly, when Program A issues subsequent calls with a *conversation_ID* of Y, CPI Communications knows that these calls apply to the conversation with Program C.

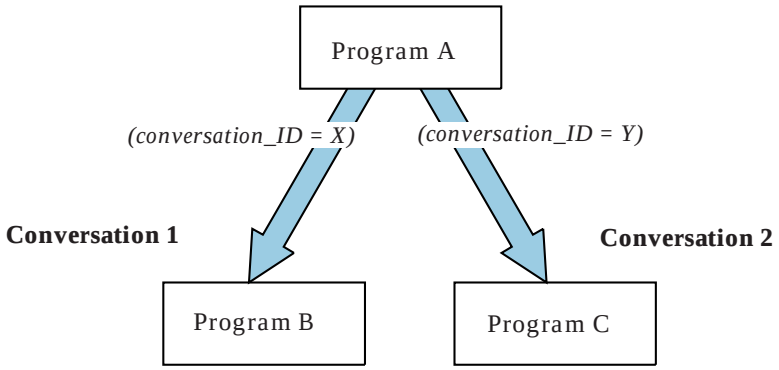


Figure 3-4 Program Using Multiple Outbound CPI Communications Conversations

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.

3.7.1.3 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 3-5, where Programs D and E have both chosen to initiate conversations with the same partner, Program S.

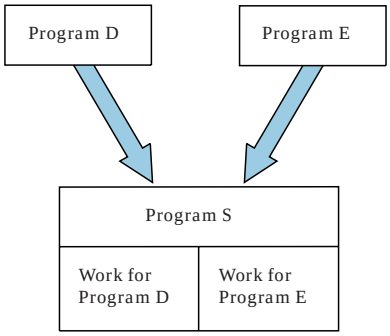


Figure 3-5 Program Using Multiple Inbound CPI Communications Conversations

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 Section 3.10 on page 43 for a more detailed discussion. See Section 4.3.7 on page 82 and Section 4.3.8 on page 84 for examples using these calls.

3.8 Conversation Characteristics

As discussed previously, 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 3-2 on page 30 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 upper-case 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 Section 1.3 on page 3.

3.8.1 Modifying and Viewing Characteristics

In the example in Section 3.7 on page 26, 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 these calls are discussed in the individual call descriptions in Chapter 5.

Note: As already stated, CPI Communications maintains conversation characteristics on a per-conversation basis. Changes to a characteristic 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 5.

Table 3-2 Characteristics and their Default Values

Name of Characteristic	Initialize_Conversation sets it to:	Accept_Conversation sets it to:
AE_qualifier	The application-entity-qualifier from side information referenced by *sym_dest_name*. If a blank *sym_dest_name* was specified, *AE_qualifier* is the null string.	For an OSI TP CRM, the initiating *AE_qualifier* received on the conversation startup request. For an LU 6.2 CRM, the null string.
AE_qualifier_length	The length of *AE_qualifier*. If a blank *sym_dest_name* was specified, *AE_qualifier_length* is 0.	The length of *AE_qualifier*.
AE_qualifier_format	The format of *AE_qualifier*. If a blank *sym_dest_name* was specified, *AE_qualifier_format* will not be meaningful.	For an OSI TP CRM, the format of *AE_qualifier*. For an LU 6.2 CRM, *AE_qualifier_format* is not set.
allocate_confirm	CM_ALLOCATE_NO_CONFIRM	Not applicable.
AP_title	The application-process-title from side information referenced by *sym_dest_name*. If a blank *sym_dest_name* was specified, *AP_title* is the null string.	For an OSI TP CRM, the initiating *AP_title* received on the conversation startup request. For an LU 6.2 CRM, the null string.
AP_title_length	The length of *AP_title*. If a blank *sym_dest_name* was specified, *AP_title_length* is 0.	The length of *AP_title*.
AP_title_format	The format of *AP_title*. If a blank *sym_dest_name* was specified, *AP_title_format* will not be meaningful.	For an OSI TP CRM, the format of *AP_title*. For an LU 6.2 CRM, *AP_title_format* is not set.
application_context_name	The application context name for UDT as defined by ISO/IEC 10026-6 for OSI TP (unstructured data transfer). This application context name is represented as 1.0.10026.6.2. If a different application context name is specified in the side information referenced by *sym_dest_name*, it will be the initial value.	For an OSI TP CRM, the initiating *application_context_name* received on the conversation startup request. For an LU 6.2 CRM, the null string.
application_context_name_length	The length of *application_context_name*. If a blank *sym_dest_name* was specified, *application_context_name* is 0.	The length of *application_context_name*.
begin_transaction	CM_BEGIN_IMPLICIT	Not applicable.
confirmation_urgency	CM_CONFIRMATION_URGENT	CM_CONFIRMATION_URGENT

Name of Characteristic	Initialize_For_Incoming sets it to:	Accept_Incoming sets it to:
AE_qualifier	Not set.	For an OSI TP CRM, the initiating *AE_qualifier* received on the conversation startup request. For an LU 6.2 CRM, the null string.
AE_qualifier_length	Not set.	The length of *AE_qualifier*.
AE_qualifier_format	Not set.	For an OSI TP CRM, the format of *AE_qualifier*. For an LU 6.2 CRM, *AE_qualifier_format* is not set.
allocate_confirm	Not applicable.	Not applicable.
AP_title	Not set.	For an OSI TP CRM, the initiating *AP_title* received on the conversation startup request. For an LU 6.2 CRM, the null string.
AP_title_length	Not set.	The length of *AP_title*.
AP_title_format	Not set.	For an OSI TP CRM, the format of *AP_title*. For an LU 6.2 CRM, *AP_title_format_* is not set.
application_context_name	Not set.	For an OSI TP CRM, the initiating *application_context_name* received on the conversation startup request. For an LU 6.2 CRM, the null string.
application_context_name_length	Not set.	The length of *application_context_name*.
begin_transaction	Not applicable.	Not applicable.
confirmation_urgency	CM_CONFIRMATION_URGENT	Not changed by Accept_Incoming.

Name of Characteristic	Initialize_Conversation sets it to:	Accept_Conversation sets it to:
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* is CM_SECURITY_SAME.	Not applicable.
conversation_state	CM_INITIALIZE_STATE	For half-duplex conversations, CM_RECEIVE_STATE. For full-duplex conversations, CM_SEND_RECEIVE_STATE.
conversation_type	CM_MAPPED_CONVERSATION	CM_MAPPED_CONVERSATION if the CRM type is OSI TP, or the value received on the conversation startup request if the CRM type is LU 6.2.
deallocate_type	CM_DEALLOCATE_SYNC_LEVEL	CM_DEALLOCATE_SYNC_LEVEL
error_direction	CM_RECEIVE_ERROR	CM_RECEIVE_ERROR
fill	CM_FILL_LL	CM_FILL_LL
initialization_data	Null	The value received on the conversation startup request.
initialization_data_length	0	The length of the initialization data received on the conversation startup request.
join_transaction	Not set.	CM_JOIN_IMPLICIT
log_data	Null	Null
log_data_length	0	0
mode_name	The mode name from side information referenced by *sym_dest_name*. If a blank *sym_dest_name* was specified, *mode_name* is the null string.	The mode name for the logical connection 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* is 0.	The length of *mode_name*.
partner_LU_name	The partner LU name from side information referenced by *sym_dest_name*. If a blank *sym_dest_name* was specified, *partner_LU_name* is a single blank.	For an LU 6.2 CRM, the partner LU name for the logical connection on which the conversation startup request arrived. For an OSI TP CRM, *partner_LU_name* is a single blank.
partner_LU_name_length	The length of *partner_LU_name*. If a blank *sym_dest_name* was specified, *partner_LU_name_length* is 1.	The length of *partner_LU_name*.

Name of Characteristic	Initialize_For_Incoming sets it to:	Accept_Incoming sets it to:
conversation_security_type	Not applicable.	Not applicable.
conversation_state	CM_INITIALIZE_INCOMING_STATE	For half-duplex conversations, CM_RECEIVE_STATE. For full-duplex conversations, CM_SEND_RECEIVE_STATE.
conversation_type	CM_MAPPED_CONVERSATION	The value received on the conversation startup request if the CRM type is LU 6.2. Not changed if the CRM type is OSI TP.
deallocate_type	CM_DEALLOCATE_SYNC_LEVEL	Not changed by Accept_Incoming.
error_direction	CM_RECEIVE_ERROR	Not changed by Accept_Incoming.
fill	CM_FILL_LL	Not changed by Accept_Incoming.
initialization_data	Null	The value received on the conversation startup request.
initialization_data_length	0	The length of the initialization data received on the conversation startup request.
join_transaction	CM_JOIN_IMPLICIT	Not changed by Accept_Incoming.
log_data	Null	Not changed by Accept_Incoming.
log_data_length	0	Not changed by Accept_Incoming.
mode_name	Not set.	The mode name for the logical connection on which the conversation startup request arrived.
mode_name_length	Not set.	The length of *mode_name*.
partner_LU_name	Not set.	For an LU 6.2 CRM, the partner LU name for the logical connection on which the conversation startup request arrived. For an OSI TP CRM, *partner_LU_name* is a single blank.
partner_LU_name_length	Not set.	The length of *partner_LU_name*.

Name of Characteristic	Initialize_Conversation sets it to:	Accept_Conversation sets it to:
prepare_data_permitted	CM_PREPARE_DATA_NOT_PERMITTED	Not applicable.
prepare_to_receive_type	CM_PREP_TO_RECEIVE_SYNC_LEVEL	CM_PREP_TO_RECEIVE_SYNC_LEVEL
processing_mode	CM_BLOCKING	CM_BLOCKING
receive_type	CM_RECEIVE_AND_WAIT	CM_RECEIVE_AND_WAIT
return_control	CM_WHEN_SESSION_ALLOCATED	Not applicable.
security_password	The security password from side information referenced by *sym_dest_name*. If a blank *sym_dest_name* was specified, *security_password* is the null string.	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.
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* is the null string.	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* is 0.	The length of *security_user_ID*.
send_receive_mode	CM_HALF_DUPLEX	The value received in the conversation startup request.
send_type	CM_BUFFER_DATA	CM_BUFFER_DATA
sync_level	CM_NONE	The value received on the conversation startup request.
TP_name	The program name from side information referenced by *sym_dest_name*. If a blank *sym_dest_name* was specified, *TP_name* is a single blank.	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* is 1.	The length of *TP_name*.
transaction_control	CM_CHAINED_TRANSACTIONS	For an OSI TP CRM, the value received on the conversation startup request. For an LU 6.2 CRM, CM_CHAINED_TRANSACTIONS.

Name of Characteristic	Initialize_For_Incoming sets it to:	Accept_Incoming sets it to:
prepare_data_permitted	Not applicable.	Not applicable.
prepare_to_receive_type	CM_PREP_TO_RECEIVE_SYNC_LEVEL	Not changed by Accept_Incoming.
processing_mode	CM_BLOCKING	Not changed by Accept_Incoming.
receive_type	CM_RECEIVE_AND_WAIT	Not changed by Accept_Incoming.
return_control	Not applicable.	Not applicable.
security_password	Not applicable.	Not applicable.
security_password_length	Not applicable.	Not applicable.
security_user_ID	Not set.	The value received on the conversation startup request.
security_user_ID_length	Not set.	The length of *security_user_ID*.
send_receive_mode	Not set.	The value received in the conversation startup request.
send_type	CM_BUFFER_DATA	Not changed by Accept_Incoming.
sync_level	Not set.	The value received on the conversation startup request.
TP_name	Not set.	The value received on the conversation startup request.
TP_name_length	Not set.	The length of *TP_name*.
transaction_control	Not set.	For an OSI TP CRM, the value received on the conversation startup request. For an LU 6.2 CRM, CM_CHAINED_TRANSACTIONS.

3.8.2 Characteristic Values and CRMs

Some conversation characteristic values are meaningful only for a particular CRM type. For example, a *sync_level* value of CM_NONE paired with a *deallocate_type* value of CM_DEALLOCATE_CONFIRM has meaning only for an OSI TP CRM. On the other hand, an *error_direction* value of CM_SEND_ERROR has meaning only for an LU 6.2 CRM. These CRM-type-sensitive characteristic values and value pairs are listed below:

- For an OSI TP CRM:
  - *allocate_confirm* (CM_ALLOCATE_CONFIRM)
  - *prepared_data_permitted* (CM_PREPARE_DATA_PERMITTED)
  - *transaction_control* (CM_UNCHAINED_TRANSACTIONS)
  - *sync_level* (CM_NONE) paired with *deallocate_type* (CM_DEALLOCATE_CONFIRM)
  - *sync_level* (CM_SYNC_POINT) paired with *deallocate_type* (CM_DEALLOCATE_CONFIRM)
  - *sync_level* (CM_SYNC_POINT) paired with *deallocate_type* (CM_DEALLOCATE_FLUSH)
  - *sync_level* (CM_SYNC_POINT_NO_CONFIRM) paired with *deallocate_type* (CM_DEALLOCATE_CONFIRM)
  - *sync_level* (CM_SYNC_POINT_NO_CONFIRM) paired with *deallocate_type* (CM_DEALLOCATE_FLUSH)
  - *sync_level* (CM_SYNC_POINT_NO_CONFIRM) paired with *send_receive_mode* (CM_HALF_DUPLEX)
- For an LU 6.2 CRM:
  - *error_direction* (CM_SEND_ERROR).

CPI Communications considers a conversation to be *using a particular CRM type* if one of the following events occurs:

- The program has successfully set a characteristic value or value pair for the conversation that is meaningful only for that CRM type.
- The conversation has been allocated on that CRM type.

When the conversation is using a particular CRM type, the implications are:

- The program receives a CM_PROGRAM_PARAMETER_CHECK return code if it attempts to set any characteristic value that is meaningful only for a different CRM type.

  For example, suppose a program has successfully set the *allocate_confirm* characteristic on a conversation to CM_ALLOCATE_CONFIRM, using the Set_Allocate_Confirm call. CPI Communications now considers this conversation to be using an OSI TP CRM. If the program then issues a Set_Error_Direction call on the conversation with *error_direction* set to CM_SEND_ERROR, the program receives a CM_PROGRAM_PARAMETER_CHECK return code.

  Details on this type of checking are provided in the individual call descriptions in Chapter 5.

- The conversation is only allocated on that CRM type.

  If a conversation is using one CRM type and complete destination information is available for both CRM types, the Allocate call tries to establish a logical connection using only the destination information for the CRM type being used.

3.8.3 Characteristic Values and Send-Receive Modes

Table 3-3 lists the values of conversation characteristics that are not applicable to full-duplex conversations. Table 3-4 lists the values of conversation characteristics that are not applicable to half-duplex conversations.

Characteristic Name	Inapplicable Values
confirmation_urgency	all values
deallocate_type	CM_DEALLOCATE_CONFIRM (for conversations using an LU 6.2 CRM only)
error_direction	all values
prepare_to_receive_type	all values
processing_mode	all values
send_type	CM_SEND_AND_CONFIRM CM_SEND_AND_PREP_TO_RECEIVE
sync_level	CM_CONFIRM CM_SYNC_POINT

Table 3-3 Conversation Characteristic Values that Cannot be Set for Full-duplex

Characteristic Name	Inapplicable Values
sync_level	CM_SYNC_POINT_NO_CONFIRM (for conversations using an LU 6.2 CRM only)

Table 3-4 Conversation Characteristic Values that Cannot be Set for Half-duplex

On a conversation with a particular send-receive mode:

- The program receives a CM_PROGRAM_PARAMETER_CHECK if it attempts to set any characteristic value that is not applicable to that send-receive mode.
- The Set_Send_Receive_Mode call returns with CM_PROGRAM_PARAMETER_CHECK if a previously set *deallocate_type*, *send_type* or *sync_level* characteristic value is not applicable to the send-receive mode specified on the call.

The following calls cannot be issued on full-duplex conversations and will receive a CM_PROGRAM_PARAMETER_CHECK return code on a full-duplex conversation:

- Confirm
- Confirmed (for conversations using an LU 6.2 CRM only)
- Prepare_To_Receive
- Request_To_Send
- Test_Request_To_Send_Received
- Set_Prepare_To_Receive_Type
- Set_Error_Direction
- Set_Confirmation_Urgency
- Set_Processing_Mode.

3.8.4 Characteristic Values and Resource Recovery Interfaces

As described in Section 3.14 on page 51, CPI Communications can be used in conjunction with different resource recovery interfaces; namely the TX (Transaction Demarcation) and the SAA resource recovery interfaces. The conversation characteristic *join_transaction* is meaningful only when used with the TX (Transaction Demarcation) interface.

3.8.5 Automatic Conversion of Characteristics

Some conversation characteristics affect only the function of the local 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 CRM and, thus, affect both ends of the conversation. For example, the local CRM transmits the *TP_name* characteristic to the remote CRM as part of the conversation startup process.

When an LU 6.2 CRM is used, CPI Communications requires 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. When an OSI TP CRM is used, the transfer syntax is negotiated by the underlying support. CPI Communications automatically converts these characteristics to the transfer syntax when they are used as parameters on CPI Communications calls.

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:

AE_qualifier
: Specified on the Extract_AE_Qualifier and Set_AE_Qualifier calls.

AP_title
: Specified on the Extract_AP_Title and Set_AP_Title calls.

application_context_name
: Specified on the Extract_Application_Context_Name and Set_Application_Context_Name calls.

initialization_data
Specified on the Extract_Initialization_Data and Set_Initialization_Data 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.

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.

TP_name
Specified on the Set_TP_Name and Extract_TP_Name calls.

3.9 Concurrent Operations

CPI Communications provides for concurrent call operations (multiple call operations in progress simultaneously) on a conversation by grouping calls in logical associations or *conversation queues*. Calls associated with one queue are processed independently of calls associated with other queues or with no queue. Table 3-5 shows the different conversation queues and calls associated with them.

The send-receive mode of the conversation determines what queues are available for the conversation. Table 3-5 shows the send-receive modes for which the conversation queues are available.

A program may initiate concurrent operations by using multiple program threads on systems with multi-threading support. (See Section 3.9.1.) Alternatively, a program may use queue-level non-blocking support to regain control when a call operation on a queue cannot complete immediately. The call operation remains in progress. The program may issue a call associated with another queue or perform other processing. Queue-level non-blocking is described in Section 3.10.2 on page 44.

Only one call operation is allowed to be in progress on a given conversation queue at a time. If a program issues a call associated with a queue that has a previous call operation still in progress, the later call returns with the CM_OPERATION_NOT_ACCEPTED return code.

3.9.1 Use of Multiple Program Threads

While CPI Communications itself does not provide multi-threading support, some implementations are designed to work with multi-threading support in the base operating system and to allow multi-threaded programs to use CPI Communications. On such a system, a program may create separate threads to initiate concurrent operations on a conversation. For example, a program may create separate threads to handle the send and receive operations on a full-duplex conversation, where the Send-Data and Receive calls are associated with the Send and Receive queues, respectively. Each thread's operations proceed independently; in particular, the sending thread may continue to send data to the partner program while the receiving thread is waiting for a Receive call to complete.

It is the responsibility of the program to ensure that action taken by one thread does not interfere with action taken by another thread. For example, unexpected results may occur if two threads issue calls associated with the same queue, or if one thread modifies the value of a conversation characteristic that affects the processing of a call issued by another thread.

Table 3-5 Conversation Queues: Associated Calls and Send-Receive Modes

Conversation Queue	CPI Communications Calls	Send-Receive Mode
Initialization	Accept_Incoming Allocate Set_AE_Qualifier Set_Allocate_Confirm Set_AP_Title Set_Application_Context_Name Set_Conversation_Security_Password Set_Conversation_Security_Type Set_Conversation_Security_User_ID Set_Conversation_Type Set_Initialization_Data	Half-duplex and full-duplex

Conversation Queue	CPI Communications Calls	Send-Receive Mode
	Set_Join_Transaction Set_Mode_Name Set_Partner_LU_Name Set_Return_Control Set_Send_Receive_Mode Set_Sync_Level Set_Transaction_Control Set_TP_Name	
Send	Confirmed Deallocate Deferred_Deallocate Flush Include_Partner_In_Transaction Prepare Send_Data Send_Error Set_Deallocate_Type Set_Log_Data Set_Prepare_Data_Permitted Set_Send_Type	Full-duplex
Receive	Receive Set_Fill Set_Receive_Type	Full-duplex
Send-Receive	Confirm Confirmed Deallocate Deferred_Deallocate Flush Include_Partner_In_Transaction Prepare Prepare_To_Receive Receive Send_Data Send_Error Set_Confirmation_Urgency Set_Deallocate_Type Set_Error_Direction Set_Fill Set_Log_Data Set_Prepare_Data_Permitted Set_Prepare_To_Receive_Type Set_Receive_Type Set_Send_Type	Half-duplex
Expedited-Send	Request_To_Send (Half-duplex only) Send_Expedited_Data	Half-duplex and full-duplex
Expedited-Receive	Receive_Expedited_Data	Half-duplex and full-duplex
Determined by the queue named on the call	Set_Queue_Callback_Function Set_Queue_Processing_Mode	Half-duplex and full-duplex

Note: The following calls are not associated with any queue.

Accept_Conversation
Cancel_Conversation
Convert_Incoming
Convert_Outgoing
Extract_*
Initialize_Conversation
Initialize_For_Incoming
Release_Local_TP_Name
Set_Begin_Transaction
Set_Join_Transaction
Set_Processing_Mode
Specify_Local_TP_Name
Test_Request_To_Send_Received
Wait_For_Conversation
Wait_For_Completion.

3.10 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.

Note: This section describes non-blocking operations for a single-threaded program, but similar considerations apply to a program issuing CPI Communications calls on multiple threads. Specifically, only the thread that issues a call is blocked if the call is processed in blocking mode and cannot complete immediately. When the program uses non-blocking support, control is returned to the calling thread if the call operation cannot complete immediately. That thread may then perform other processing, including issuing calls on the same conversation.

When the non-blocking processing mode applies to a call and the call operation cannot complete immediately, CPI Communications returns control to the program with a return code of CM_OPERATION_INCOMPLETE. The call operation remains in progress as an *outstanding operation*, and the program is allowed to perform other processing. The following calls can return the CM_OPERATION_INCOMPLETE return code:

Accept_Incoming
Allocate
Confirm
Confirmed
Deallocate
Deferred_Deallocate
Flush
Include_Partner_In_Transaction
Prepare
Prepare_To_Receive
Receive
Receive_Expedited_Data
Request_To_Send
Send_Data
Send_Error
Send_Expedited_Data

Table 3-6 Calls Returning CM_OPERATION_INCOMPLETE

CPI Communications provides two levels of support for programs using the non-blocking processing mode: *conversation* level and *queue* level. These are discussed in the sections below. Until a program chooses a non-blocking level for a conversation, all calls on the conversation are processed in blocking mode.

Note: A program may choose to use conversation-level non-blocking or queue-level non-blocking, but not both, on a given conversation. Once set, the level of non-blocking used on a conversation cannot be changed. Additionally, the level of non-blocking used depends on the *send_receive_mode* characteristic. The program can choose to use either level of non-blocking support on a half-duplex conversation. However, the program can use only queue-level non-blocking on a full-duplex conversation.

3.10.1 Conversation-level Non-blocking

Conversation-level non-blocking allows only one outstanding operation on a conversation at a time. The program chooses conversation-level non-blocking by issuing the Set_Processing_Mode (CMSPM) call to set the *processing_mode* conversation characteristic. The *processing_mode* characteristic indicates whether subsequent calls on the conversation are to be processed in blocking or non-blocking mode.

If *processing_mode* is set to CM_NON_BLOCKING and a call receives the CM_OPERATION_INCOMPLETE return code, the call operation becomes an outstanding operation on the conversation. The program must issue the Wait_For_Conversation (CMWAIT) call to determine when the outstanding operation is completed and to retrieve the return code for that operation. CPI Communications keeps track of all conversations using conversation-level non-blocking and having an outstanding operation, and responds to a subsequent Wait_For_Conversation call with the conversation identifier of one of those conversations when the operation on it completes.

With conversation-level non-blocking, only one call operation is allowed to be in progress on the conversation at a time. Any call (except Cancel_Conversation) issued on the conversation while the previous call operation is still in progress gets 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.

3.10.2 Queue-level Non-blocking

In contrast to conversation-level non-blocking, queue-level non-blocking allows more than one outstanding operation per conversation. CPI Communications allows programs using queue-level non-blocking to have one outstanding operation per queue simultaneously.

With queue-level non-blocking, the processing mode is set on a queue basis. The program chooses queue-level non-blocking by issuing the Set_Queue_Processing_Mode (CMSQPM) or Set_Queue_Callback_Function (CMSQCF) call to set the queue processing mode for a specified queue. Until the program sets the processing mode for a queue, all calls associated with that queue are processed in blocking mode. Calls not associated with any queue are processed in blocking mode and are always completed before control is returned to the program.

3.10.2.1 Working with Wait Facility

When using the Set_Queue_Processing_Mode call, the program manages multiple outstanding operations with *outstanding-operation identifiers*, or OOIDs. CPI Communications creates and maintains a unique OOID for each queue. Additionally, a program may choose to associate a *user field* with an outstanding operation. The user field is provided as an aid to programming, and might be used to contain, for example, the address of a data structure with return parameters for an outstanding operation.

When a call receives the CM_OPERATION_INCOMPLETE return code, the call operation becomes an outstanding operation on the conversation queue with which the call is associated. The program must issue the Wait_For_Completion call to wait for the operation to complete and to obtain the corresponding OOID and user field.

Here is a scenario of how a program might use queue-level non-blocking on a full-duplex conversation:

1. The program uses the Set_Queue_Processing_Mode call to set the processing mode for the Send queue to non-blocking. It also supplies a user field that contains the address of a parameter list for the Send_Data call and receives back an OOID from CPI Communications that is unique to the Send queue.
2. The program next uses the Set_Queue_Processing_Mode call to set the processing mode for the Receive queue to non-blocking. This time it supplies a user field that contains the address of a parameter list for the Receive call. It receives back an OOID from CPI Communications that is unique to the Receive queue.
3. The program issues a Send_Data call, which returns CM_OPERATION_INCOMPLETE, followed by a Receive call, which also returns CM_OPERATION_INCOMPLETE. If the program attempted to issue another call associated with either queue, it would receive a CM_OPERATION_NOT_ACCEPTED return code because there can be only one outstanding operation at a time per queue. Note that when a call on a conversation receives a CM_OPERATION_INCOMPLETE return code, the conversation does not change state.
4. The program can now issue a Wait_For_Completion call to wait for both outstanding operations at the same time. It does this by specifying a list of OOIDs for the outstanding operations it wants to wait on. When the Wait_For_Completion call returns, it indicates which operations have completed (if any), along with a list of user fields. The state transition triggered by the completed operation occurs when the Wait_For_Completion call completes.
5. The program uses the parameter-list address in the user field to determine the results of a given completed operation.

3.10.2.2 Using Callback Function

An alternative use of queue-level non-blocking is to establish a *callback function* and a user field for the conversation queue using the Set_Queue_Callback_Function (CMSQCF) call. When an outstanding operation completes, the program is interrupted and the callback function is called (passing the user field and call ID as input data). See *Set_Queue_Callback_Function (CMSQCF)* on page 297 for details. When the callback function returns, the program continues from where it was interrupted.

3.10.3 Cancel Outstanding Operations

The program may use the Cancel_Conversation (CMCANC) call to end a conversation. The call terminates all the call operations in progress on the conversation. The terminated call operations do not return a return code.

3.11 Conversation Security

Many systems control access to system resources through security parameters associated with a request for access to those resources. In particular, a CRM 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.

The conversation startup request contains one of the following forms of access security information:

- no access security information
- the user ID of the user on whose behalf access to the remote program is requested
- the user ID and a password for the user on whose behalf access to the remote program is requested
- authentication tokens for the user on whose behalf access to the remote program is requested.

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.
- When the program allocates a conversation with *conversation_security* set to CM_SECURITY_SAME, the security parameters for the program are used to generate the access security information.

Note: An OSI TP CRM cannot support conversation security because conversation security is not supported by existing OSI TP standards.

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

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

3.12 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. See the **APPLICATION USAGE** sections of *Convert_Incoming (CMCNVI)* on page 143 and *Convert_Outgoing (CMCNVO)* on page 145.

3.13 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:

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. This state is applicable only for half-duplex conversations.

Receive The program is able to receive data on this conversation. This state is applicable only for half-duplex conversations.

Send-Pending The program has received both data and send control on the same Receive call. See Section 4.3.6 on page 80 for a discussion of the **Send-Pending** state. This state is applicable only for half-duplex conversations.

Confirm A confirmation request has been received on this conversation; that is, the remote program issued either a Confirm call or a Send_Data call with *Send_Type* set to CM_SEND_AND_CONFIRM, 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. This state is applicable only for half-duplex conversations.

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 either the *sync_level* is CM_CONFIRM, or the *sync_level* is CM_SYNC_POINT and the conversation is not currently included in a transaction. After responding with Confirmed, the local program's end of the conversation enters **Send** state. This state is applicable only for half-duplex conversations.

Confirm-Deallocate A confirmation request and deallocation notification have both been received on this conversation. For a half-duplex conversation, the remote program issued a Deallocate call in one of the following situations:

— *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_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT, and the conversation is not currently included in a transaction.

For a full-duplex conversation, the remote program issued a Deallocate call with the *deallocate_type* set to CM_DEALLOCATE_CONFIRM.

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.

Send-Receive The program can send and receive data on this conversation. This state is applicable only for full-duplex conversations.

Send-Only The program can only send data on this conversation. This state is applicable only for full-duplex conversations.

Receive-Only The program can only receive data on this conversation. This state is applicable only for full-duplex conversations.

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 half-duplex 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 a *sync_level* of CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM:

Defer-Receive (for half-duplex conversations only)
Defer-Deallocate
Prepared
Sync-Point
Sync-Point-Send (for half-duplex conversations only)
Sync-Point-Deallocate.

Section 3.14 on page 51 discusses synchronization point processing and describes these additional states.

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

3.14 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 specification uses the TX (Transaction Demarcation) interface in its examples that illustrate how CPI Communications works with resource recovery interfaces.

For information on using the TX (Transaction Demarcation) interface, see the referenced **TX** (Transaction Demarcation) specification.

For information about performing synchronization point processing with the SAA resource recovery interface, see the referenced SAA CPI Resource Recovery specification 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 or CM_SYNC_POINT_NO_CONFIRM. This kind of conversation is called a *protected conversation*. In this specification the terms *protected conversation* and *conversation included in a transaction* are synonymous. Note that with unchained transactions, a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM can be protected or non-protected. (See Section 3.14.5 on page 58.)

3.14.1 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. When using the TX (Transaction Demarcation) interface, the program must demarcate the first sync point by issuing a *tx_begin*( ) call. When using the SAA resource recovery interface, the first sync point is automatically set for a program when it successfully accesses its first protected resource. When the first sync point is set, the program is placed in transaction mode. The program uses a resource recovery interface's commit call such as *tx_commit*( ) to establish a new sync point or a resource recovery interface's backout call such as *tx_rollback*( ) to return to a previous sync point. The processing and the changes to resources that occur between one sync point and the next are collectively referred to as a *transaction* or a *logical unit of work*.

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

CPI Communications does not provide any means to manage different transactions in one program. Therefore server programs accepting multiple protected conversations can be realised only on systems with multi-threading support.

3.14.2 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 tables show 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, or a Prepare call with the *prepare_data_permitted* conversation characteristic set to CM_PREPARE_DATA_NOT_PERMITTED, while its end of the conversation was in **Send** or **Send-Pending** state. The local program's end of the conversation is in **Sync-Point** state and is 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, or a Prepare call with the *prepare_data_permitted* conversation characteristic set to CM_PREPARE_DATA_NOT_PERMITTED, 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 is placed in **Send** state once the local program issues a successful commit call.
CM_TAKE_COMMIT_DEALLOCATE	The partner program issued a commit call, or a Prepare call with the *prepare_data_permitted* conversation characteristic set to CM_PREPARE_DATA_NOT_PERMITTED, either while its end of the conversation was in **Defer-Deallocate** state or after issuing a Deferred_Deallocate call. The local program's end of the conversation is in **Sync-Point-Deallocate** state and is placed in **Reset** state once the local program issues a successful commit call.
CM_TAKE_COMMIT_DATA_OK	The partner program issued a Prepare call with the *prepare_data_permitted* conversation characteristic set to CM_PREPARE_DATA_PERMITTED while its end of the conversation was in **Send** or **Send-Pending** state. The local program's end of the conversation is in **Sync-Point** state and is placed back in **Receive** state once the local program issues a successful commit call.
CM_TAKE_COMMIT_SEND_DATA_OK	The partner program issued a Prepare call with the *prepare_data_permitted* conversation characteristic set to CM_PREPARE_DATA_PERMITTED 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 is placed in **Send** state once the local program issues a successful commit call.
CM_TAKE_COMMIT_DEALLOC_DATA_OK	The partner program issued a Prepare call with the *prepare_data_permitted* conversation characteristic set to CM_PREPARE_DATA_PERMITTED, either while its end of the conversation was in **Defer-Deallocate** state or after issuing a Deferred_Deallocate call. The local program's end of the conversation is in **Sync-Point-Deallocate** state and is placed in **Reset** state once the local program issues a successful commit call.

Table 3-7 Possible Take-commit Notifications for Half-duplex Conversations

status_received Value	Conditions for Receipt
CM_TAKE_COMMIT	The partner program issued a commit call, or the conversation is using an LU 6.2 CRM and the partner program issued a Prepare call, while its end of the conversation was in **Send-Receive** state. The local program's end of the conversation is in **Sync-Point** state and is placed back in **Send-Receive** state once the local program issues a successful commit call.
CM_TAKE_COMMIT_DEALLOCATE	The partner program issued a commit call, or the conversation is using an LU 6.2 CRM and the partner program issued a Prepare call, either while its end of the conversation was in **Defer-Deallocate** state or after issuing a Deferred_Deallocate call. The local program's end of the conversation is in **Sync-Point-Deallocate** state and is placed in **Reset** state once the local program issues a successful commit call.
CM_TAKE_COMMIT_DATA_OK	The conversation is using an OSI TP CRM, and the partner program issued a Prepare call while its end of the conversation was in **Send-Receive** state. The local program's end of the conversation is in **Sync-Point** state and is placed back in **Send-Receive** state once the local program issues a successful commit call.
CM_TAKE_COMMIT_DEALLOC_DATA_OK	The conversation is using an OSI TP CRM, and the partner program issued a Prepare call, either while its end of the conversation was in **Defer-Deallocate** state or after issuing a Deferred_Deallocate call. The local program's end of the conversation is in **Sync-Point-Deallocate** state and is placed in **Reset** state once the local program issues a successful commit call.

Table 3-8 Possible Take-commit Notifications for Full-duplex Conversations

When the program issues a backout call, or when a system failure or a problem with a protected resource causes the TM 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_FAIL_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
Deallocate(S)[3]
Extract_Conversation_State
Flush
Prepare
Prepare_To_Receive
Receive
Send_Data
Send_Error.

3.14.3 The Backout-Required Condition

Upon receipt of a take-backout notification on a protected conversation, the program 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, possibly including multiple conversations.

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

- when CPI Communications returns a take-backout notification
- when the program issues a Cancel_Conversation call or 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 on a protected conversation, CPI Communications places the program in the **Backout-Required** condition.

When a program is placed in the **Backout-Required** condition, the program should issue a resource recovery backout call. Until it has issued a backout call, the program is unable to successfully issue any of the following CPI Communications calls for any of its protected conversations. If the program issues any of these calls, the CM_PROGRAM_STATE_CHECK return code is returned:

3. Deallocate(S) refers to a Deallocate call issued with the *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL and the *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.

Allocate
Confirm
Confirmed
Deallocate (unless *deallocate_type* is set to CM_DEALLOCATE_ABEND)
Flush
Prepare
Prepare_To_Receive
Receive
Request_To_Send
Send_Data
Send_Error
Test_Request_To_Send_Received.

Until it has issued a backout call, the program is also unable to successfully issue any of the following CPI Communications calls for any of its non-protected conversations with *transaction_control* set to CM_UNCHAINED_TRANSACTIONS and *begin_transaction* set to CM_BEGIN_IMPLICIT (see Section 3.14.5 on page 58 and Section 3.14.6 on page 59). If the program issues any of these calls, the CM_PROGRAM_STATE_CHECK return code is returned:

Allocate
Confirm
Prepare
Prepare_To_Receive
Receive
Send_Data
Send_Error.

3.14.4 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 3-9 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 3-9 Responses to Take-commit and Take-backout Notifications

Notification Received	Possible Response	Reason for Response	Result of Response 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)‡ or Cancel_Conversation	The program has detected an error condition that prevents it from continuing normal processing.	The program is placed in the **Backout-Required** condition.	The program should issue 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 TM 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)‡ or Cancel_Conversation	The program has detected an error condition that prevents it from continuing normal processing.	The program is placed in the **Backout-Required** condition.	The program should issue a resource recovery backout call.

Notes:

† If the take-commit indicator ended in *_DATA_OK, the partner may also send data before making any of the other possible responses.

‡ ''Deallocate (Abend)'' refers to the CPI Communications Deallocate call with a *deallocate_type* of CM_DEALLOCATE_ABEND.

§ The program can respond with a Send_Error call only when using a half-duplex conversation.

3.14.5 Chained and Unchained Transactions

When a program is using the TX (Transaction Demarcation) interface, it may choose when the next transaction is started after the current transaction ends. Specifically, if the TX *transaction_control* characteristic is set to:

- TX_CHAINED, a commit or rollback call ends the current transaction and immediately begins the next transaction and establishes a new sync point.
- TX_UNCHAINED, a commit or rollback call ends the current transaction but does not begin the next transaction. The program that issued the commit call to end the current transaction must issue the *tx_begin*( ) call to the TX (Transaction Demarcation) interface to start the next transaction and to establish a new sync point.

For a conversation using an OSI TP CRM, the program that initialises a conversation may use the Set_Transaction_Control call to specify whether it wants to use chained or unchained transactions for the conversation. The remote program may determine whether chained or unchained transactions are being used for the conversation by issuing the Extract_Transaction_Control call. A conversation using an LU 6.2 CRM must use chained transactions.

For a conversation using chained transactions, if a commit or rollback call ends the current transaction and immediately begins the next transaction, the conversation is automatically included in that next transaction, so it is always a protected conversation. If the commit or rollback call does not immediately start the next transaction, the conversation is deallocated by the system, and the program is notified of the deallocation by a CM_RESOURCE_FAILURE_RETRY return code.

For a conversation using unchained transactions, when a commit or rollback call ends the current transaction, the conversation is not automatically included in the next transaction. Until the next transaction is started and the conversation is included in that transaction, the conversation is not a protected conversation, and any commit or rollback processing does not apply to that conversation. When the next transaction is started and the program requests that the partner program join the transaction, the conversation becomes protected again and therefore is included in that transaction.

The TX *transaction_control* characteristic and the CPI Communications *transaction_control* conversation characteristic are independent. There are four possible combinations:

- TX_CHAINED and CM_CHAINED_TRANSACTIONS
- TX_CHAINED and CM_UNCHAINED_TRANSACTIONS
- TX_UNCHAINED and CM_CHAINED_TRANSACTIONS
- TX_UNCHAINED and CM_UNCHAINED_TRANSACTIONS.

3.14.6 Joining a Transaction

For a conversation using chained transactions, when the local program issues an Allocate call after setting the *sync_level* to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and the remote program issues an Accept_Conversation or Accept_Incoming call, the remote program normally joins the transaction automatically.

For a conversation using unchained transactions, when a new transaction is started, the local program has two ways of requesting that the partner join the transaction:

1. By making an *implicit* request, the local program can issue a Set_Begin_Transaction call with a *begin_transaction* value of CM_BEGIN_IMPLICIT, followed by any of the following CPI Communications calls from **Initialize**, **Send**, **Send-Pending** or **Send-Receive** states:

   Allocate
   Confirm
   Include_Partner_In_Transaction
   Prepare
   Prepare_To_Receive
   Receive
   Send_Data
   Send_Error.

   In this case, when the local program issues the second CPI Communications call, the remote program receives a *status_received* value of CM_JOIN_TRANSACTION and normally joins the transaction automatically.

   Note: If the local program is not in transaction when one of the above calls is made, the *begin_transaction* characteristic is ignored, and the partner program is not requested to join a transaction.

2. By making an *explicit* request, the local program can issue a Set_Begin_Transaction call with a *begin_transaction* value of CM_BEGIN_EXPLICIT. At this point, no indication is sent to the remote program. The remote program does not receive the CM_JOIN_TRANSACTION value until the local program issues an Include_Partner_In_Transaction call.

If the remote program receives a *status_received* value of CM_JOIN_TRANSACTION, it normally joins automatically. When using the TX (Transaction Demarcation) interface, the program can issue a *tx_info*( ) call to see whether it is in transaction mode or not.

When using the TX (Transaction Demarcation) interface in either a chained or an unchained transaction, the program can choose not to join automatically. In this case the program must issue a Set_Join_Transaction call with *join_transaction* set to CM_JOIN_EXPLICIT. This call should be issued in the **Initialize_Incoming** state, so that it has an effect at the following Accept_Incoming call. If a program wants to use CM_JOIN_EXPLICIT, it should extract the *transaction_control* characteristic after a successful Accept_Incoming call. If the value is CM_CHAINED_TRANSACTIONS, the program should join the transaction by issuing a *tx_begin*( ) call. If the value is CM_UNCHAINED_TRANSACTIONS, the program is informed with a CM_JOIN_TRANSACTION *status_received* value, if it is to join the transaction. In any case, the program may first do any local work that is not for inclusion in the remote program's transaction before joining the transaction. Instead of issuing a *tx_begin*( ) call, the program may also reject the request to join the transaction by issuing a Deallocate call with a *deallocate_type* of CM_DEALLOCATE_ABEND or a Cancel_Conversation call.

3.14.7 Superior and Subordinate Programs

The concept of superior and subordinate programs applies only for conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM that are are using an OSI TP CRM.

The *superior* program is the program that initiates the conversation (using the Initialize_Conversation call). A program that issues the Accept_Conversation or Accept_Incoming call is a *subordinate* of the superior program. Figure 3-6 shows a commit tree with seven programs participating in the transaction.

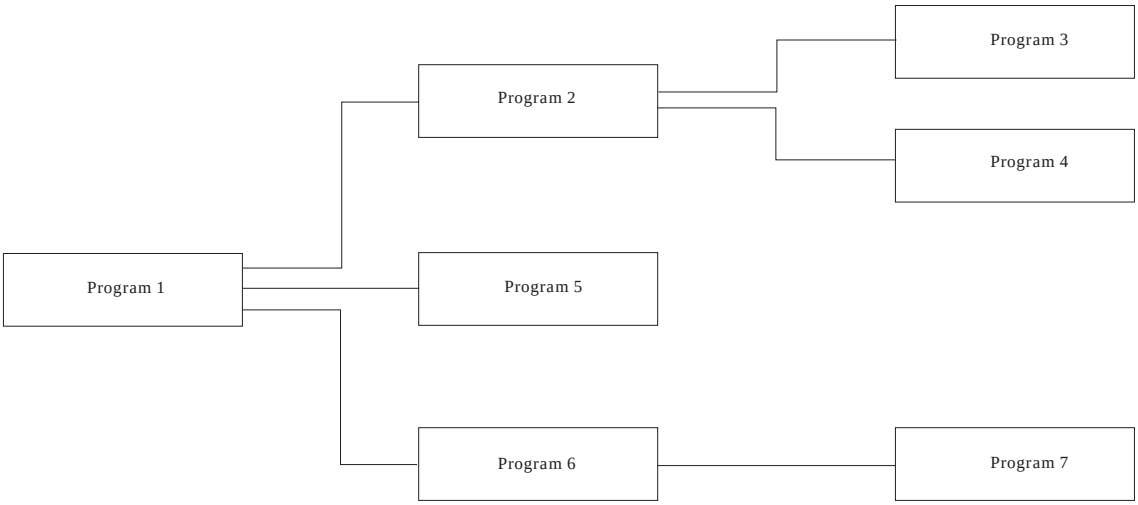


Figure 3-6 Commit Tree with Program 1 as Root and Superior

In this example, Program 1 is the superior program in its conversations with Programs 2, 5, and 6, which are all its subordinates. Similarly, Program 2 is the superior in its conversations with Programs 3 and 4, and Program 6 is the superior in its conversation with Program 7.

Only the superior program that initiated the transaction (Program 1 in this case) can issue the initial commit call to end the transaction. However, any of the superior programs in the transaction (in this example, Programs 1, 2, and 6) can issue the Deferred_Deallocate call to their subordinates (but not to their superiors).

In addition, the Include_Partner_In_Transaction, Prepare, Set_Begin_Transaction, and Set_Prepare_Data_Permitted calls may be issued only by the superior program. These calls return a CM_PROGRAM_PARAMETER_CHECK return code when they are issued by the subordinate. The Set_Join_Transaction call may be issued only by the subordinate program. This call returns a CM_PROGRAM_PARAMETER_CHECK return code when it is issued by the superior.

3.14.8 Additional CPI Communications States

In addition to the conversation states described in Section 3.13 on page 49, the following states are required when a program uses a protected CPI Communications conversation:

Defer-Receive The local program's end of the conversation enters **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 CM_SYNC_POINT_NO_CONFIRM, 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 or CM_SYNC_POINT_NO_CONFIRM.

Defer-Receive state is applicable for half-duplex conversations only.

Defer-Deallocate The local program has requested that the conversation be deallocated after a commit operation has completed; that is, the conversation is included in a transaction, and the program has issued a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, or it has 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 or CM_SYNC_POINT_NO_CONFIRM. The conversation is not deallocated until a successful commit operation takes place.

Prepared The local program has issued a Prepare call to request that the remote program prepare its resources for commitment.

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 or CM_TAKE_COMMIT_DATA_OK. After a successful commit operation, a half-duplex conversation returns to **Receive** state, while a full-duplex conversation returns to **Send-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 or CM_TAKE_COMMIT_SEND_DATA_OK. After a successful commit operation, the conversation is placed in **Send** state.

Sync-Point-Send state is applicable for half-duplex conversations only.

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 or CM_TAKE_COMMIT_DEALLOC_DATA_OK. After a successful commit operation, the conversation is deallocated and placed in **Reset** state.

3.14.9 Valid States for Resource Recovery Calls

A program must ensure that there are no outstanding operations on its protected conversations before issuing a resource recovery call. If a resource recovery call is issued while there is an outstanding operation on a protected conversation, the program receives from the resource recovery interface a return code indicating an error. All protected conversations must be in one of the following states for the program to issue a commit call:

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

If a commit call is issued from any other conversation state, the program receives from the resource recovery interface a return code indicating an error. The program can also receive an error return code if the conversation was in **Send** or **Send-Receive** 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.

3.14.10 TX Extensions for CPI Communications

If the subordinate program uses the TX (Transaction Demarcation) interface and the *join_transaction* characteristic has the value CM_JOIN_IMPLICIT, the state table in Chapter 7 of the referenced **TX** (Transaction Demarcation) specification changes in the following way:

- An incoming conversation request causes an implicit *tx_set_transaction-control*( ) call in the following way:
  - — If the incoming conversation request sets the CPI-C *transaction_control* characteristic to CM_CHAINED_TRANSACTIONS, the TX *transaction_control* characteristic changes to TX_CHAINED.
  - — If the incoming conversation request sets the CPI-C *transaction_control* characteristic to CM_UNCHAINED_TRANSACTIONS, the TX *transaction_control* characteristic changes to TX_UNCHAINED.
- An incoming conversation request which sets the CPI-C *transaction_control* characteristic to CM_CHAINED_TRANSACTIONS causes an implicit *tx_begin*( ) call. This causes implicit TX state changes.
- A *status_received* value of CM_JOIN_TRANSACTION causes an implicit *tx_begin*( ) call. This causes implicit TX state changes.

The program can use the *tx_info*( ) call to determine whether it is in transaction mode and to determine the value of the TX *transaction_control* characteristic.

Chapter 4

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 divided into two sections:

- starter-set flows; see Section 4.2 on page 64
- advanced-function flows; see Section 4.3 on page 70.

In addition to these sample flows, a simple COBOL application using CPI Communications calls is provided in Appendix F.

4.1 Interpreting the Flow Diagrams

In the flow diagrams shown in this chapter (for example, Figure 4-1 on page 67), 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 5.

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 5.

4.2 Starter-set Flows

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

- Section 4.2.1 on page 65 demonstrates a flow of data in only one direction (only the initiating program sends data).
- Section 4.2.2 on page 68 describes a bidirectional flow of data (the initiating program sends data and then allows the partner program to send data).

4.2.1 Data Flow in One Direction

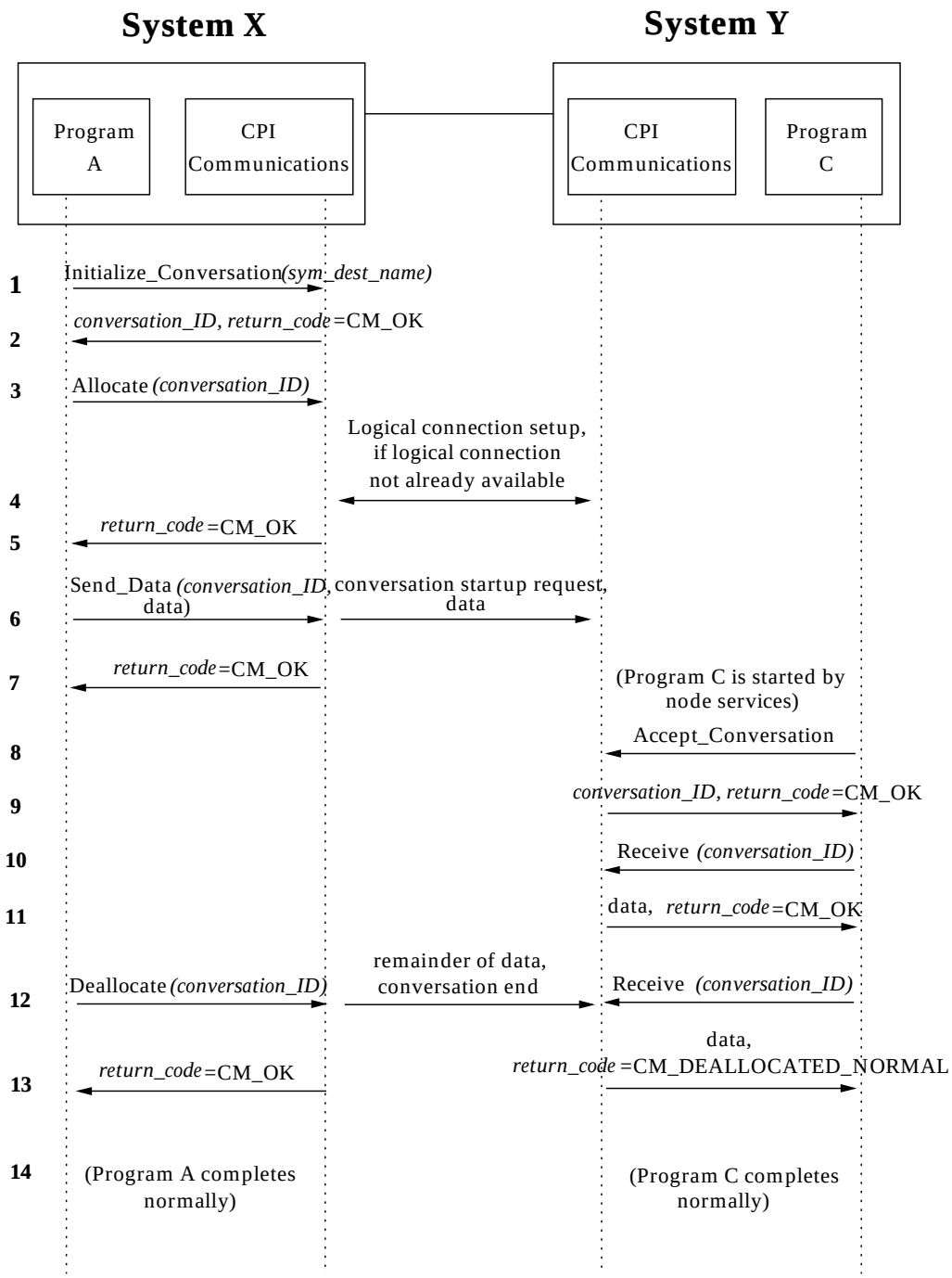
Figure 4-1 on page 67 shows an example of a conversation where the flow of data is in only one direction.

The steps shown in Figure 4-1 on page 67 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 195, 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 130) using the *conversation_ID* previously assigned by the Initialize_Conversation call.
4	If a logical connection 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 logical connections available) are not discussed. See *Allocate (CMALLC)* on page 130 for more information about the error conditions that can result.
6 and 7	Program A sends data with the Send_Data call (described in *Send_Data (CMSEND)* on page 230) 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.

Step	Description
	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 Section 4.3.1 on page 71.
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 125 for details). Program C, like Program A in Step 2, receives a unique *conversation_ID* that it uses in all future CPI Communications calls for that particular conversation. As discussed in Section 3.8 on page 29, 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 208). 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 147) 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 4-1 Data Flow in One Direction



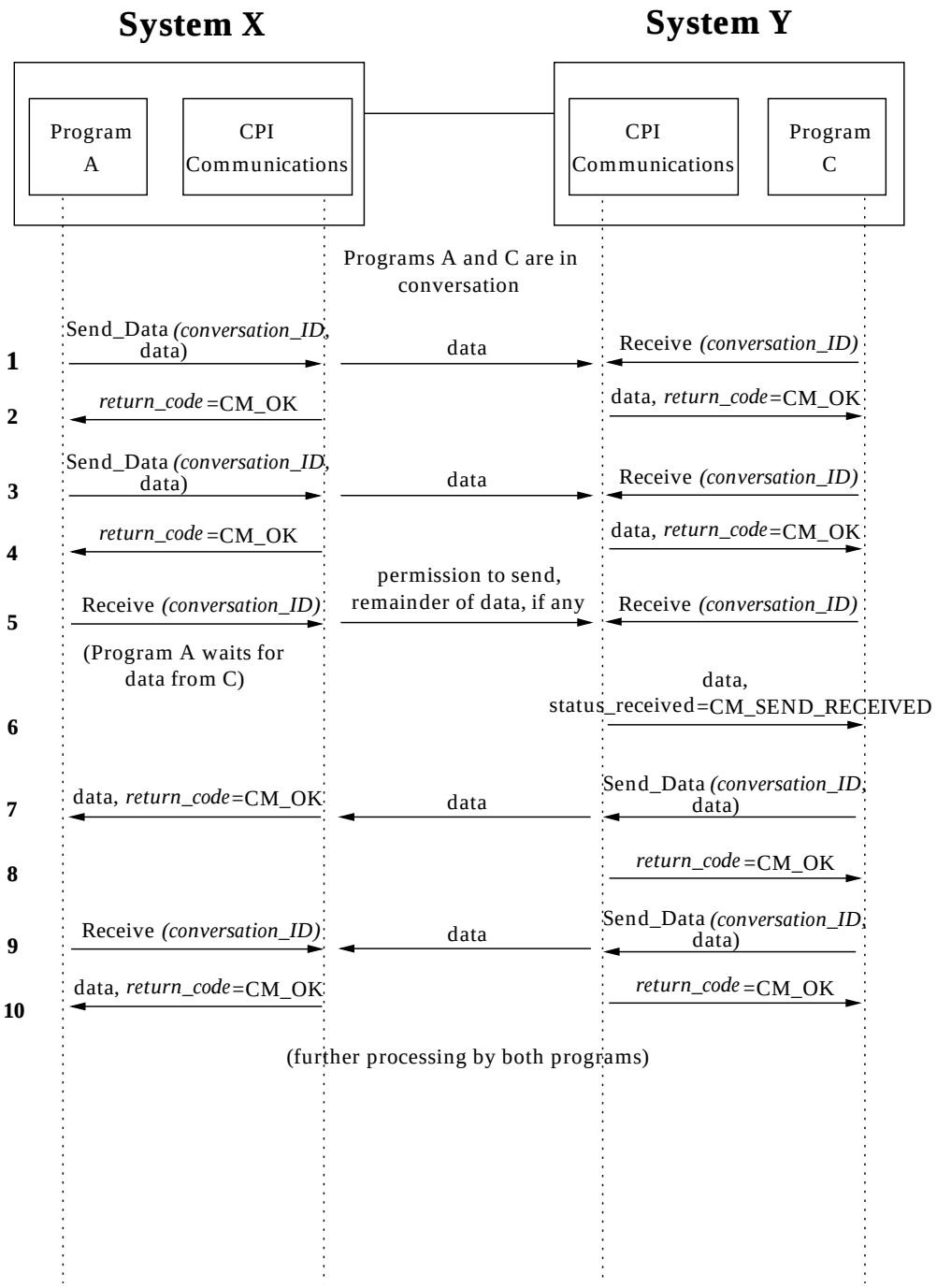
4.2.2 Data Flow in Both Directions

Figure 4-2 on page 69 shows a conversation in which the flow of data is in both directions. It describes how two programs using starter-set calls can initiate a change of control over who is sending the data.

The steps shown in Figure 4-2 on page 69 are:

Step	Description
1 to 4 inclusive	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 208, 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 Section 4.3.2 on page 72 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 to 10 inclusive	Programs A and C continue processing. Program C sends data and Program A receives the data.

Figure 4-2 Data Flow in Both Directions



4.3 Advanced-function Flows

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

- Section 4.3.2 on page 72 shows how to use the Prepare_To_Receive call to change the direction of the data flow on a half-duplex conversation.
- Section 4.3.3 on page 74 shows how to use the Confirm and Confirmed calls to validate data reception on a half-duplex conversation. The Flush call is also shown.
- Section 4.3.4 on page 76 shows how to use the Request_To_Send call to request a change in the direction of the data flow on a half-duplex conversation.
- Section 4.3.5 on page 78 shows how to use the Send_Error call to report errors in the data flow on a half-duplex conversation.
- Section 4.3.6 on page 80 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 for a Receive call on a half-duplex conversation.
- Section 4.3.7 on page 82 shows a program that uses blocking calls to accept multiple incoming half-duplex conversations.
- Section 4.3.8 on page 84 shows a program that uses non-blocking calls to accept multiple incoming half-duplex conversations.
- Section 4.3.9 on page 86 describes how a full-duplex conversation is established.
- Section 4.3.10 on page 88 describes how a full-duplex conversation is used to send and receive data.
- Section 4.3.11 on page 90 describes how a full-duplex conversation can be terminated.
- Section 4.3.12 on page 92 describes how a program uses queue-level non-blocking.
- Section 4.3.13 on page 94 shows a program sending data on a protected half-duplex conversation and using the TX (Transaction Demarcation) interface to issue a commit instruction.
- Section 4.3.14 on page 100 shows a conversation between two programs with two chained transactions on a half-duplex conversation.
- Section 4.3.15 on page 106 shows a conversation between two programs with unchained transactions on a half-duplex conversation.
- Section 4.3.16 on page 112 shows a successful commit with a conversation state change on a half-duplex conversation.

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

4.3.1 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 any time 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 Section 4.2 on page 64. The other calls are discussed in the following examples.

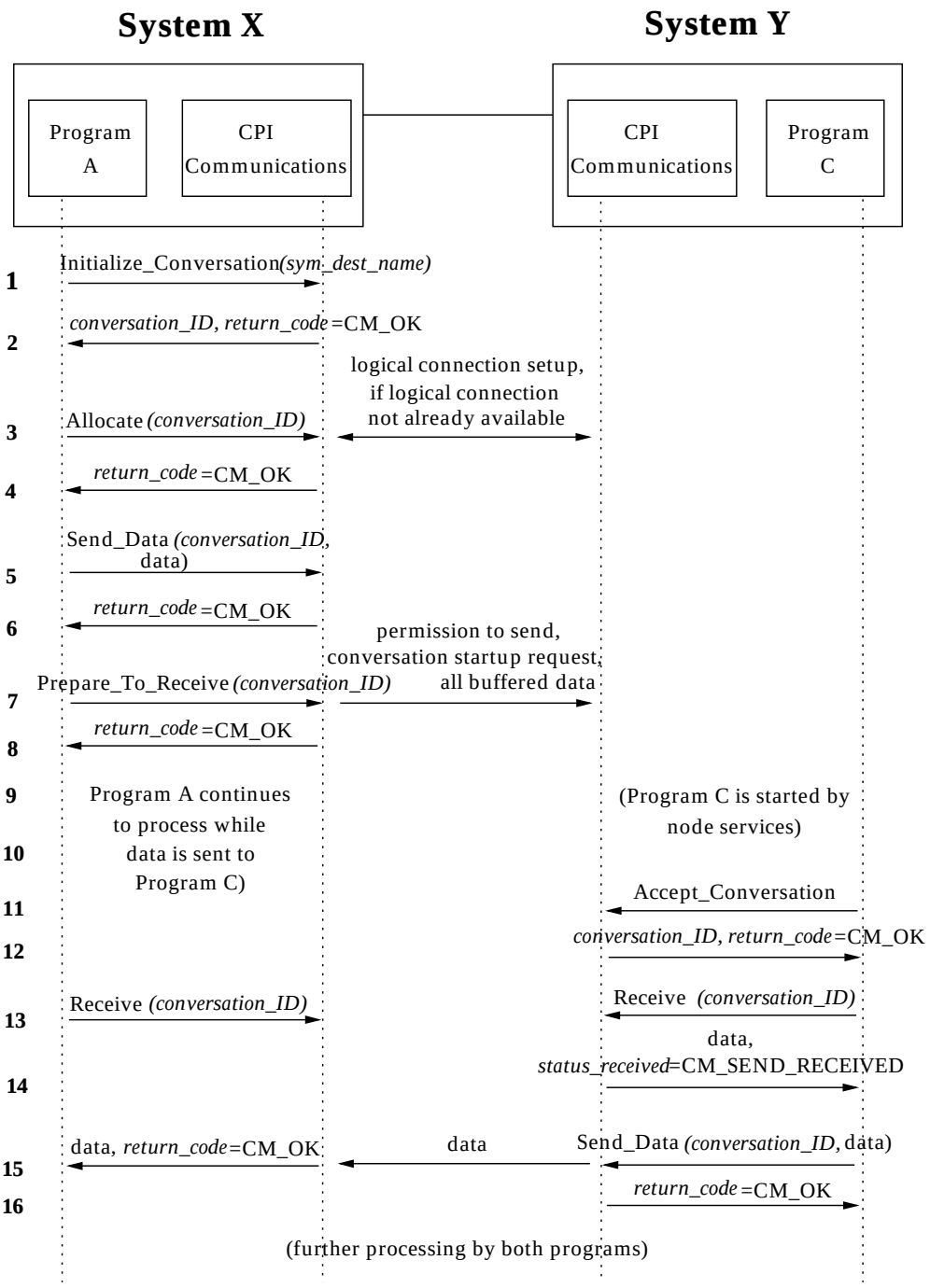
4.3.2 The Sending Program Changes the Data Flow Direction

Figure 4-3 on page 73 is a variation on the function provided by the flow shown in Section 4.2.2 on page 68. When the data flow direction changes, Program A can continue processing instead of waiting for data to arrive on this half-duplex conversation.

The steps shown in Figure 4-3 on page 73 are:

Step	Description
1 to 6 inclusive	The program begins the same as Section 4.2.1 on page 65. Program A establishes the conversation and makes the initial transmission of data.
7 to 10 inclusive	Program A makes use of an advanced-function call, Prepare_To_Receive, (described in *Prepare_To_Receive (CMPTR)* on page 202), 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 to 13 inclusive	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 now waits until data is received (Step 15).
14 to 16 inclusive	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 4-3 on page 73. For further discussion of this alternate flow, see *Set_Receive_Type (CMSRT)* on page 304 and *Receive (CMRCV)* on page 208.

Figure 4-3 Sending Program Changes the Data Flow Direction



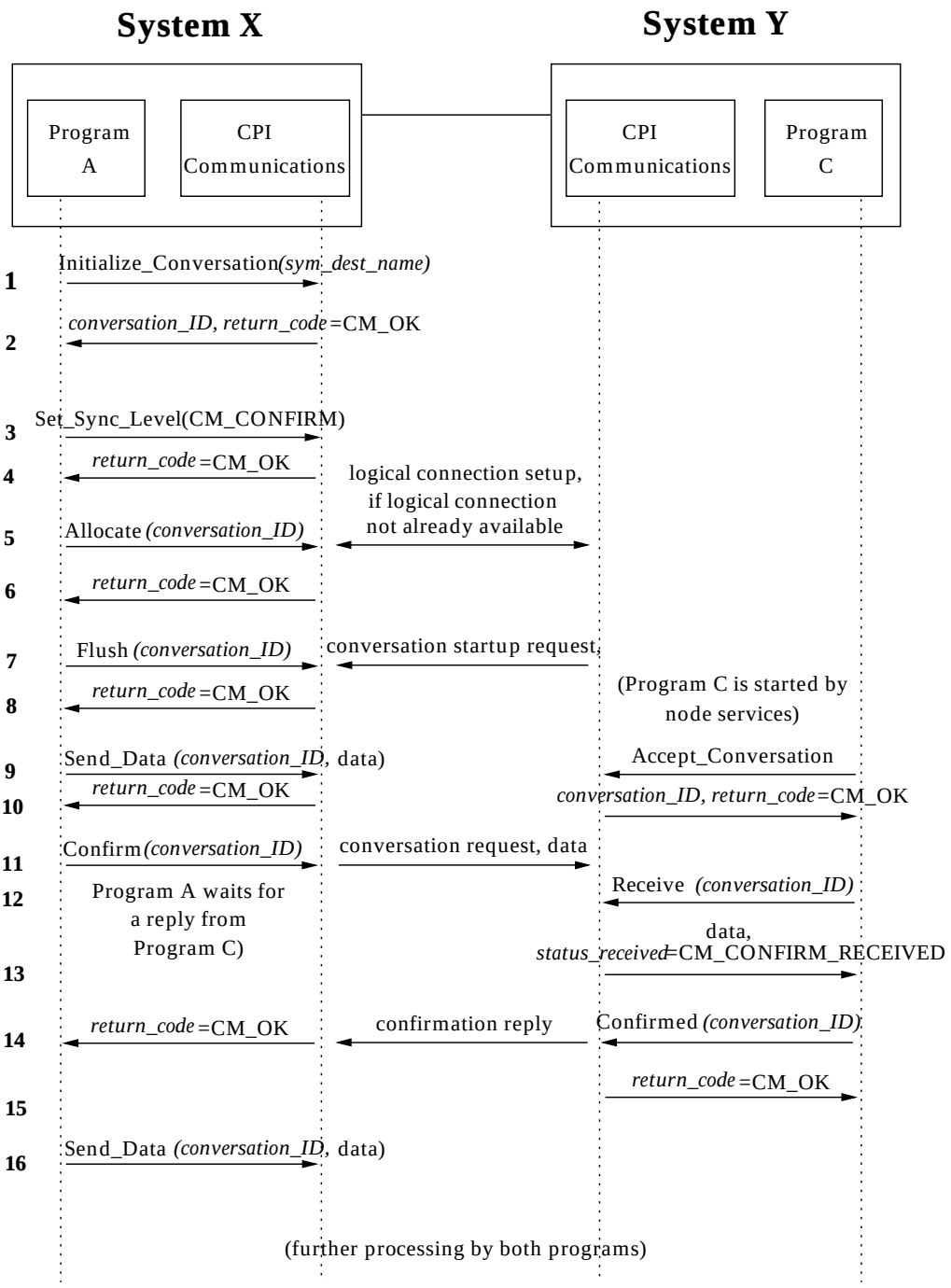
4.3.3 Validation and Confirmation of Data Reception

Figure 4-4 on page 75 shows how a program can use the Confirm and Confirmed calls on a half-duplex conversation to verify receipt of its sent data. The Flush call is also shown.

The steps shown in Figure 4-4 on page 75 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 311 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 190) 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 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 141), 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 4-4 Validation and Confirmation of Data Reception



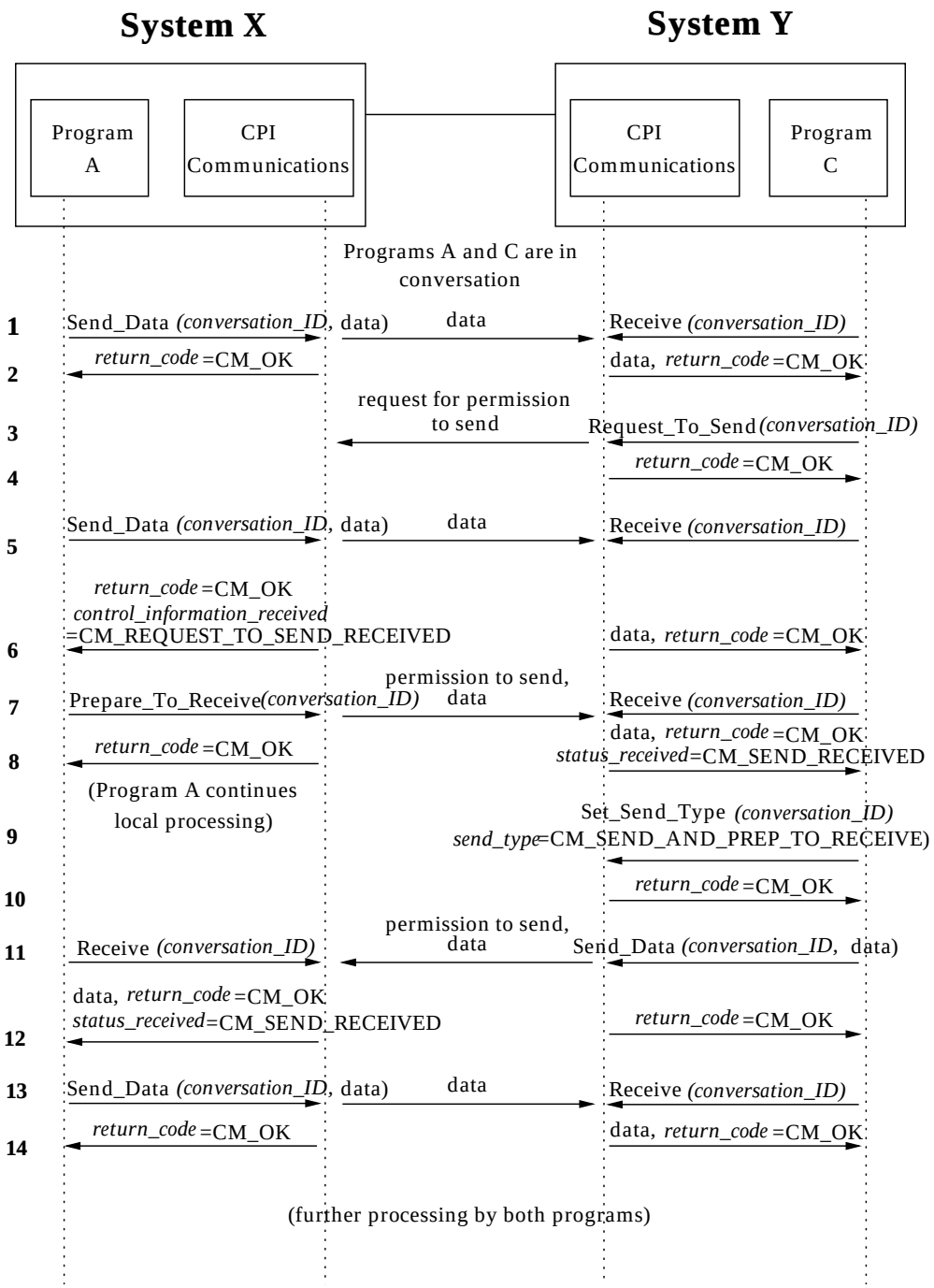
4.3.4 The Receiving Program Changes the Data Flow Direction

Figure 4-5 on page 77 shows how a program on the receiving side of a half-duplex conversation can request a change in the direction of data flow with the Request_To_Send call. (See *Request_To_Send (CMRTS)* on page 227 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 4-5 on page 77 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 is 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 *control_information_received* set equal to CM_REQ_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 Section 4.3.2 on page 72 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 returns 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 4-5 Confirmation of Data



4.3.5 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 honour a request-to-send request. Figure 4-6 on page 79 shows how to use the Send_Error call in these situations.

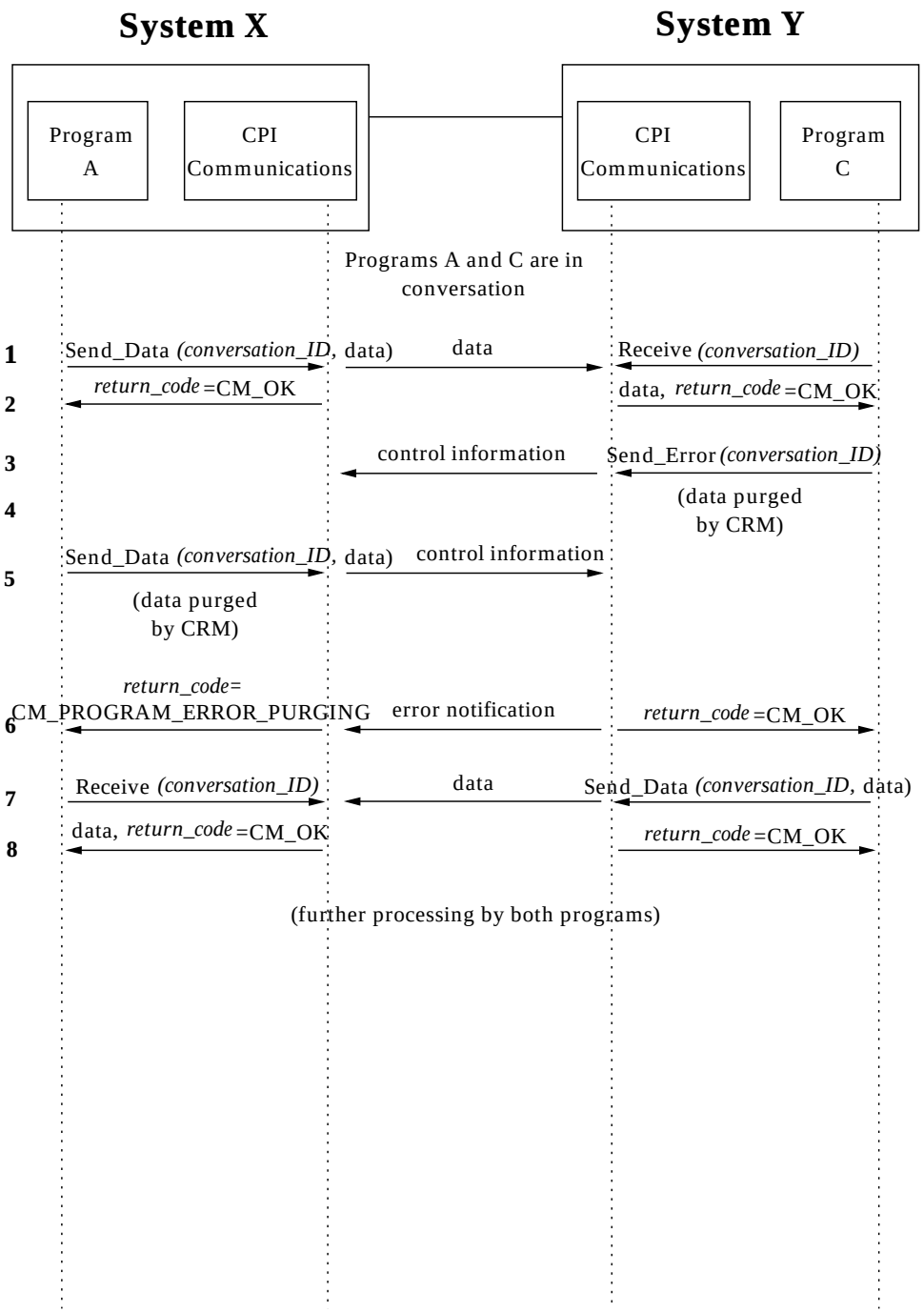
The programs are using a half-duplex conversation in this example.

Note: This example describes the simplest type of error reporting, an error found while receiving data. Section 4.3.6 on page 80 describes a more complicated use of Send_Error.

The steps shown in Figure 4-6 on page 79 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 purged 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 4-6 Reporting Errors



4.3.6 Error Direction and Send-Pending State

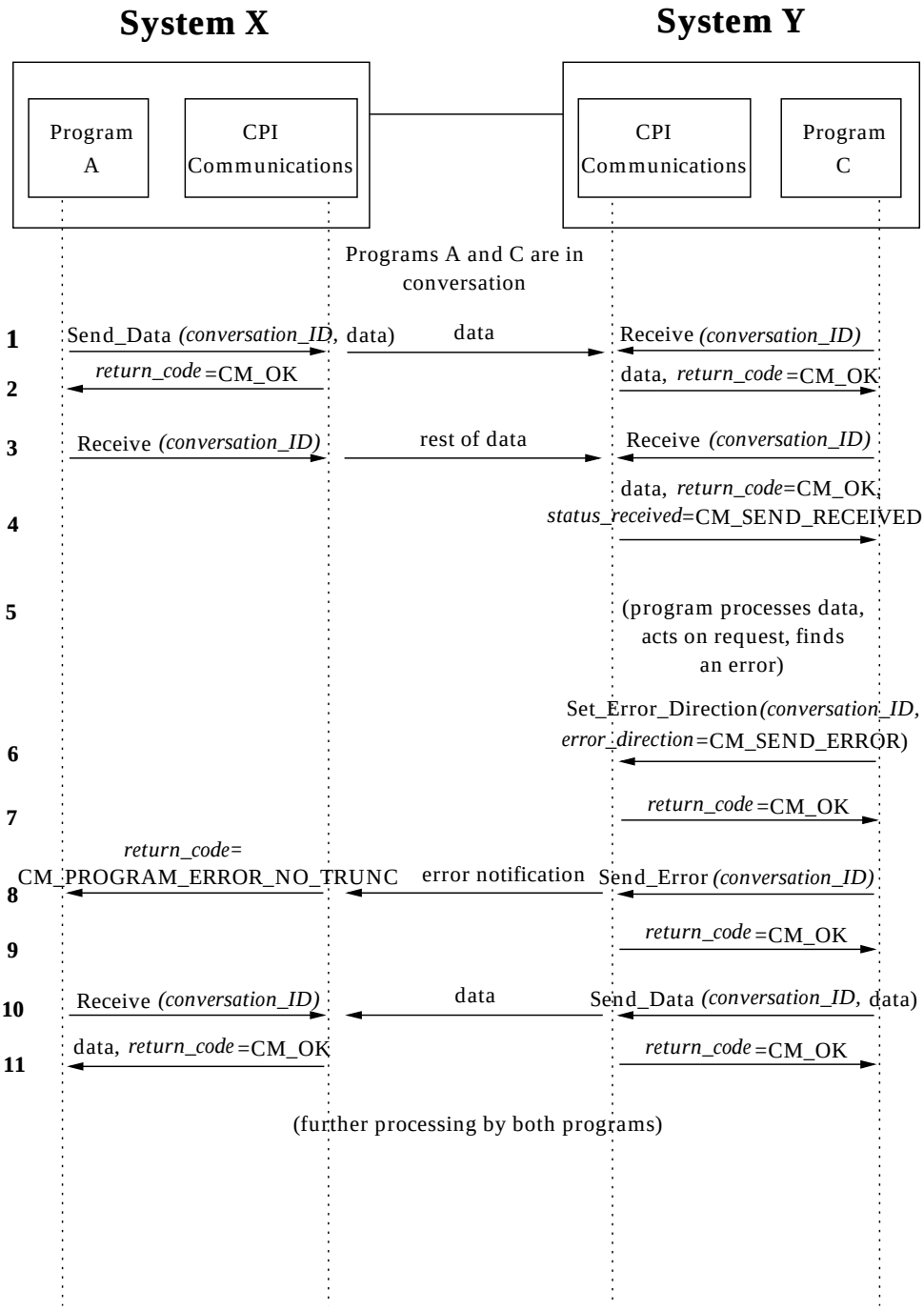
Figure 4-7 on page 81 shows how to use the **Send-Pending** state and the *error_direction* characteristic to resolve an ambiguous error condition that can occur when a program receives both a change of direction indication and data on a Receive call.

This example applies only to a half-duplex conversation using an LU 6.2 CRM.

The steps shown in Figure 4-7 on page 81 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 Section 4.3.5 on page 78). 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_ERROR_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 to 11 inclusive	Program C notifies Program A of the exact nature of the problem and both programs continue processing.

Figure 4-7 Error Direction and Send-Pending State



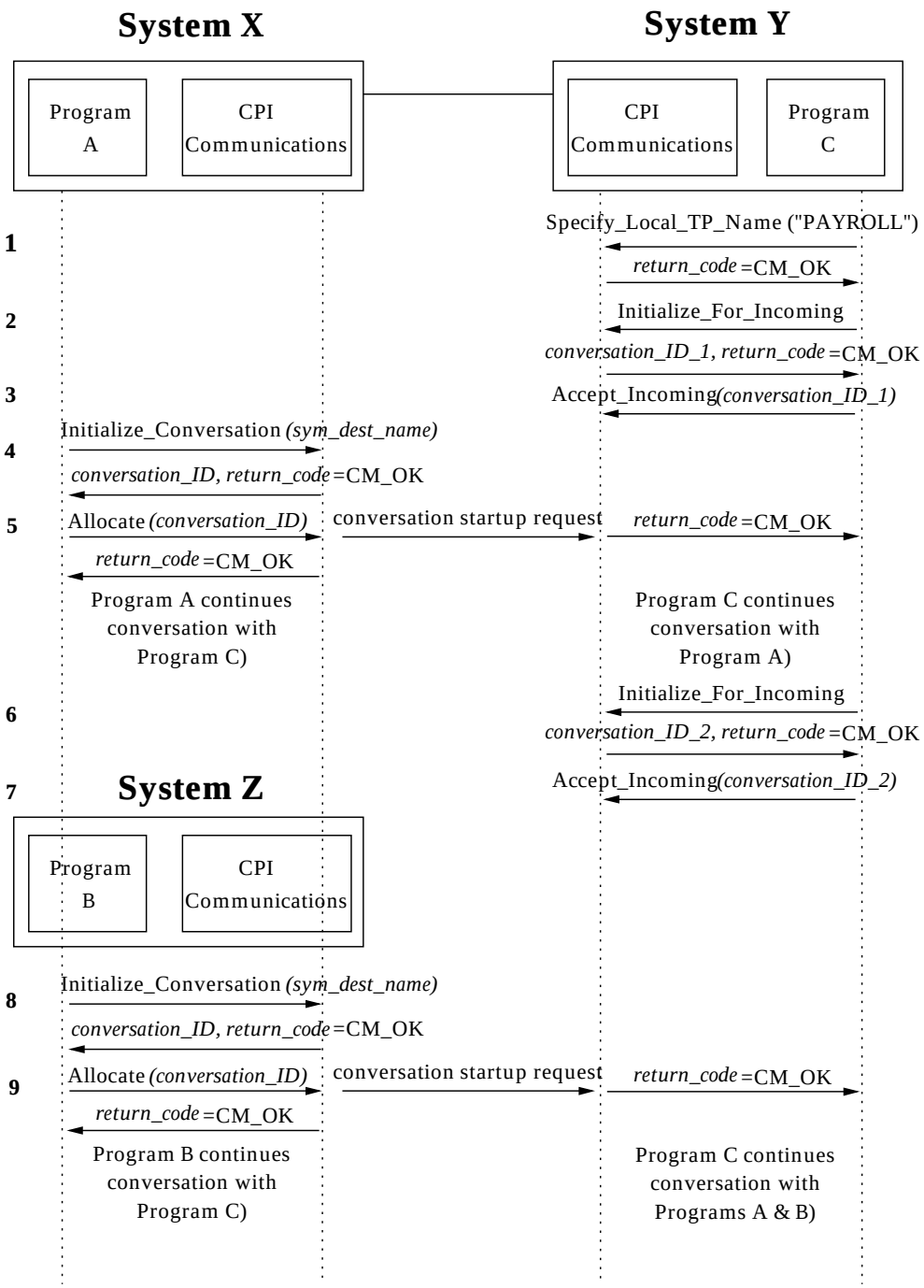
4.3.7 Multiple Conversations Using Blocking Calls

Figure 4-8 on page 83 shows an example of a program that uses blocking calls to accept multiple incoming half-duplex conversations.

The steps shown in Figure 4-8 on page 83 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 blocks 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.
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 blocks 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.

Figure 4-8 Accepting Multiple Conversations Using Blocking Calls



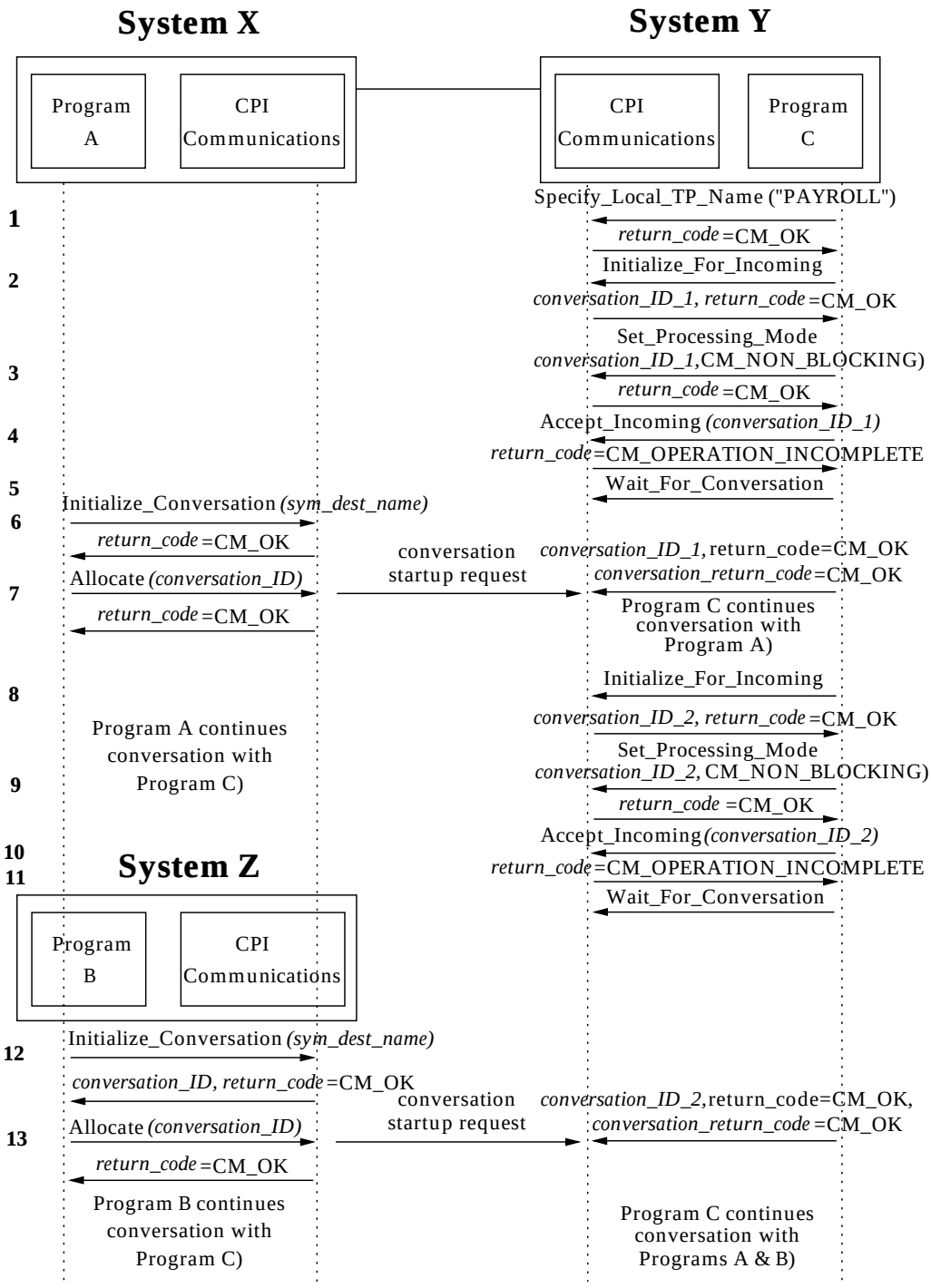
4.3.8 Multiple Conversations Using Conversation-level Non-blocking Calls

Figure 4-9 on page 85 shows an example of a program that uses conversation-level non-blocking calls to accept multiple incoming half-duplex conversations.

The steps shown in Figure 4-9 on page 85 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 is 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*.
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*.

Figure 4-9 Accepting Multiple Conversations Using Non-blocking Calls



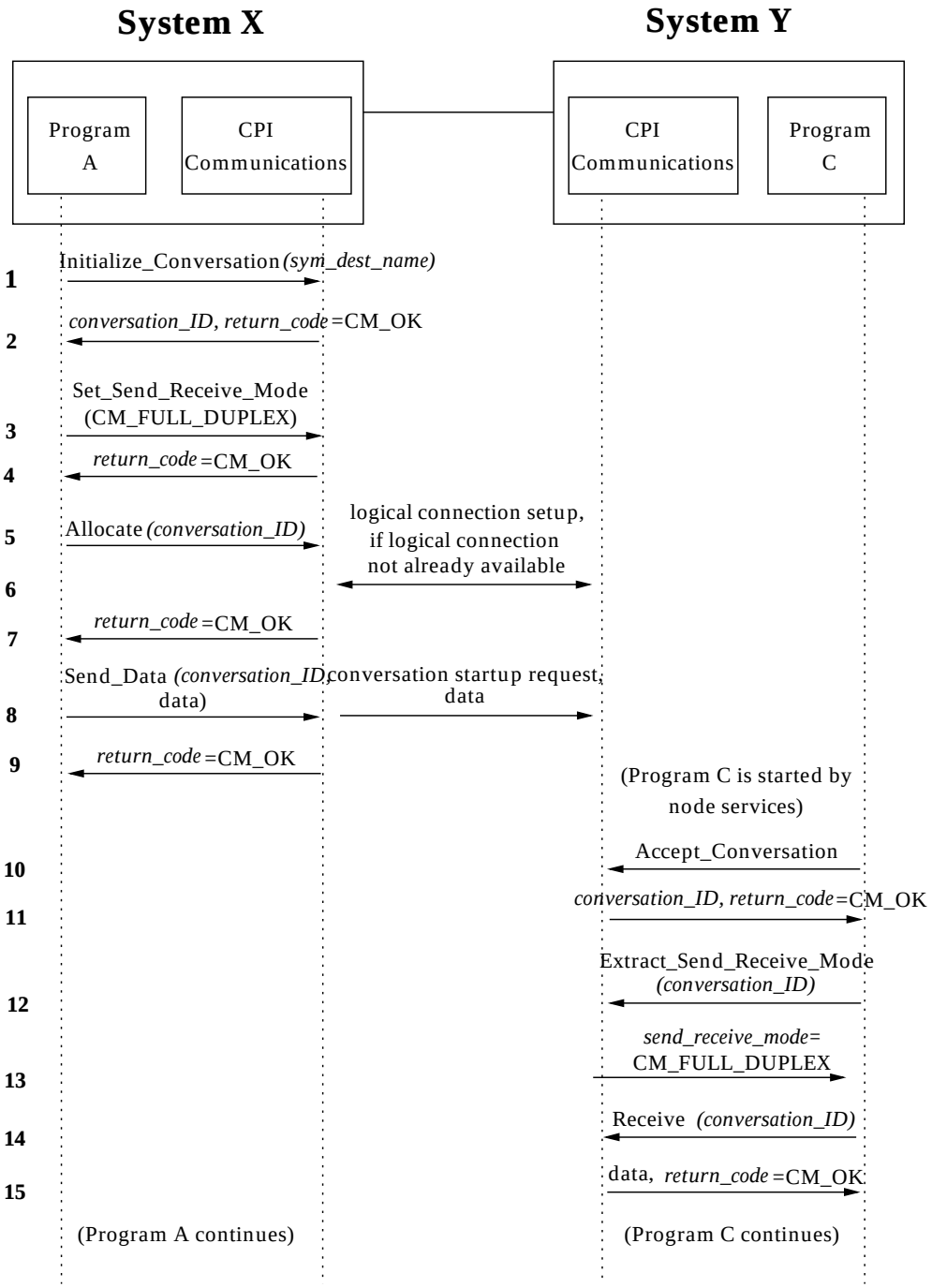
4.3.9 Establishing a Full-duplex Conversation

Figure 4-10 on page 87 is an example of how a full-duplex conversation is set up.

The steps shown in Figure 4-10 on page 87 are:

Step	Description
1 and 2	Program A initializes a conversation using the Initialize_Conversation call.
3 and 4	The default value of the *send_receive_mode* characteristic is set to CM_HALF_DUPLEX. Since the program wants to have a full-duplex conversation, it issues the Set_Send_Receive_Mode call to set the *send_receive_mode* characteristic to CM_FULL_DUPLEX.
5 to 7 inclusive	Program A allocates the full-duplex conversation. Program A's conversation state changes from **Initialize** state to **Send_Receive** state, and Program A can begin to send and receive data.
8 and 9	Program A sends data with the Send_Data call and receives a *return_code* of CM_OK. The request for a conversation is sent at this time, and it carries the *send_receive_mode*.
10 and 11	The remote system starts Program C. The conversation on Program C's side is in **Reset** state. Program C accepts the conversation, and the conversation state changes to **Send-Receive** state. Some of Program C's conversation characteristics are based on information contained in the conversation startup request. In particular, the *send_receive_mode* is set to CM_FULL_DUPLEX.
12 and 13	Program C issues Extract_Send_Receive_Mode to determine whether the conversation is half-duplex or full-duplex; the returned *send_receive_mode* value indicates that it is a full-duplex conversation.
14 and 15	Once its end of the conversation is in **Send-Receive** state, Program C begins whatever processing role it and Program A have agreed upon. In this case, Program C receives data with a Receive call.

Figure 4-10 Establishing a Full-duplex Conversation



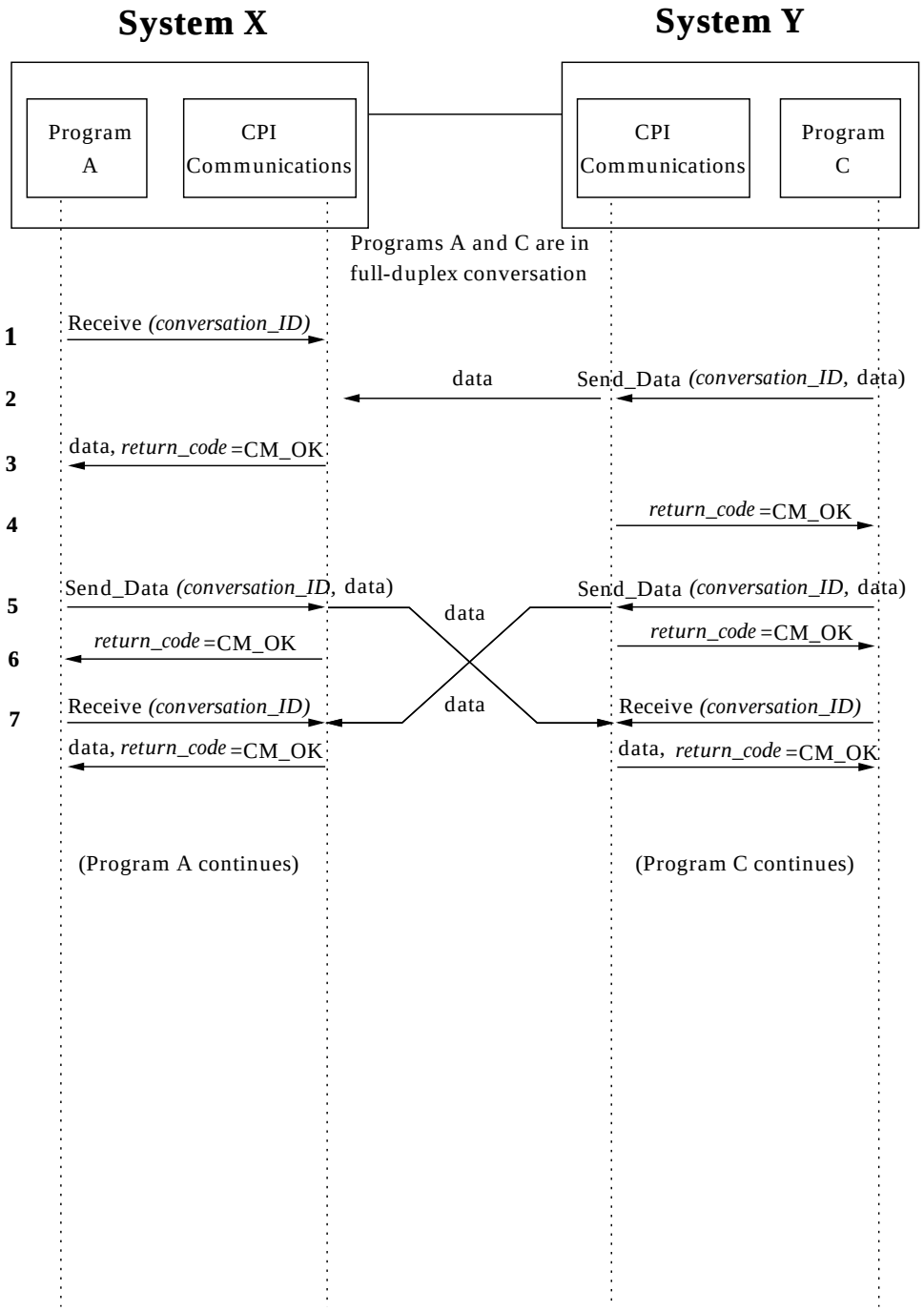
4.3.10 Using a Full-duplex Conversation

Figure 4-11 on page 89 shows an example of how a full-duplex conversation is used to send and receive data.

The steps shown in Figure 4-11 on page 89 are:

Step	Description
1	Programs A and C are in a full-duplex conversation. Both Program A's and Program C's ends of the conversation are in **Send-Receive** state. Both programs can issue a Send_Data call or a Receive call. In this example, Program A wants to receive a response to some previous request it sent to Program C, and so it issues the Receive call.
2	Program C issues a Send_Data call to send data to Program A. In this example, the data is sent to Program A right away.
3	Program A receives data from program C.
4 and 5	Both programs issue Send_Data calls, and the calls complete successfully.
6 and 7	Both programs issue Receive calls, and the Receive calls complete successfully. The state of the conversation at Program A and Program C continues to be **Send-Receive** state.

Figure 4-11 Using a Full-duplex Conversation



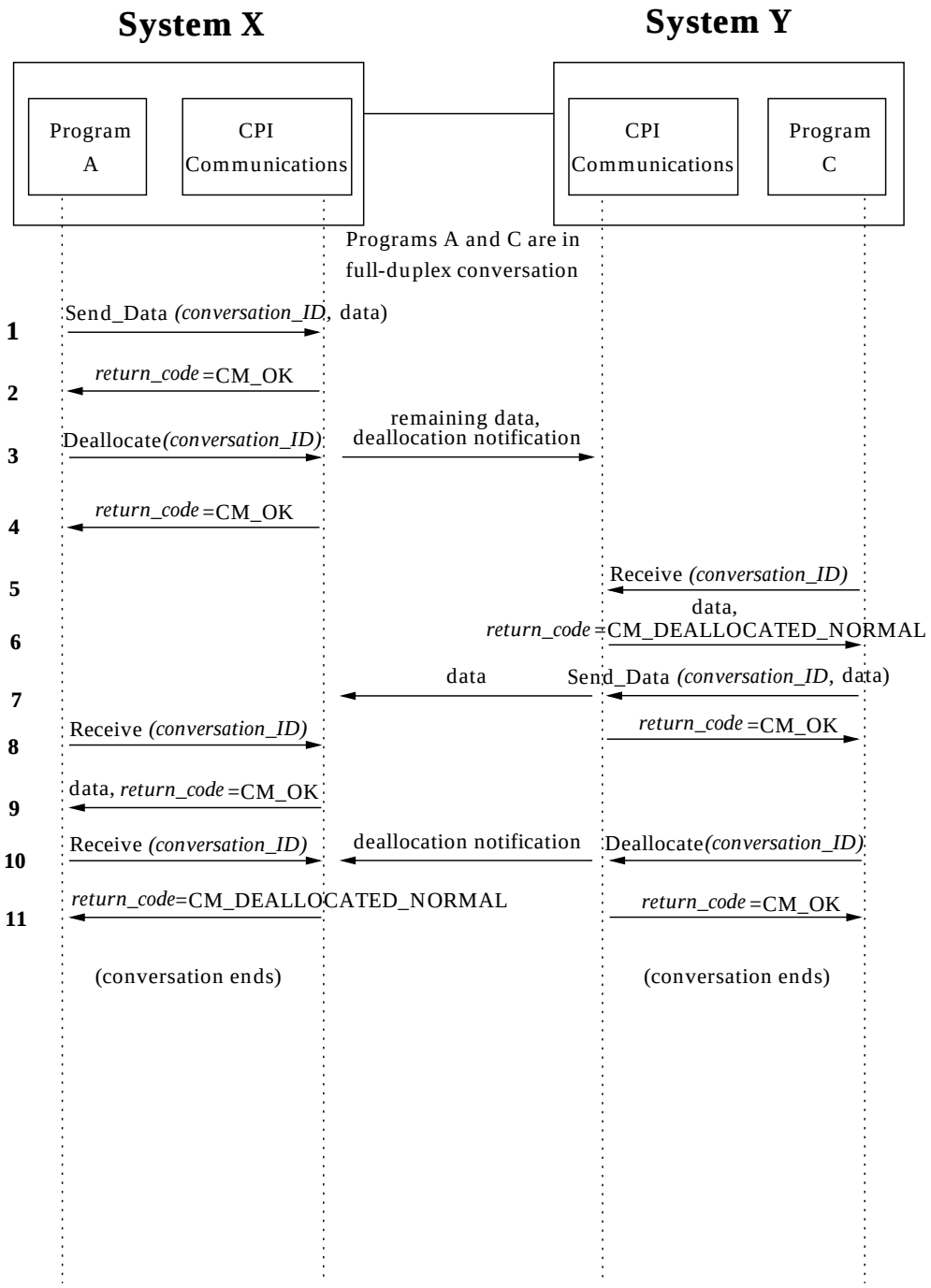
4.3.11 Terminating a Full-duplex Conversation

Figure 4-12 on page 91 shows an example of how a full-duplex conversation can be terminated.

The steps shown in Figure 4-12 on page 91 are:

Step	Description
1 and 2	The state of the conversation at Program A and Program C is **Send-Receive** state. Program A issues a Send_Data call, which completes successfully. Note that the data is not actually sent to the partner program but is buffered.
3 and 4	Program A has finished sending all data, and issues a Deallocate call. When the call completes successfully, the data in the CRM's buffers is flushed to the partner along with a deallocation notification. The conversation state at Program A's end now makes a transition to **Receive-Only** state. Program A can no longer send any data on this conversation.
5 and 6	Program C's end is in **Send-Receive** state, and Program C issues a Receive call. Program C gets back data and a return code of CM_DEALLOCATED_NORMAL. Program C's end of the conversation now enters **Send_Only** state. Program C can no longer receive any data on this conversation.
7	Program C issues a Send_Data call, and the data gets sent to Program A.
8 and 9	Program A issues a Receive call and gets data.
10	Program C's end of the conversation is in **Send-Only** state, and Program C has finished sending data. It issues a Deallocate call for the *conversation_ID*. Program A issues a Receive call to receive data.
11	Program A gets a return code of CM_DEALLOCATED_NORMAL, and its end of the conversation goes from **Receive-Only** state to **Reset** state. Program C gets a return code of CM_OK for the Deallocate call it issued earlier. Its end of the conversation goes from **Send-Only** state to **Reset** state.

Figure 4-12 Terminating a Full-duplex Conversation



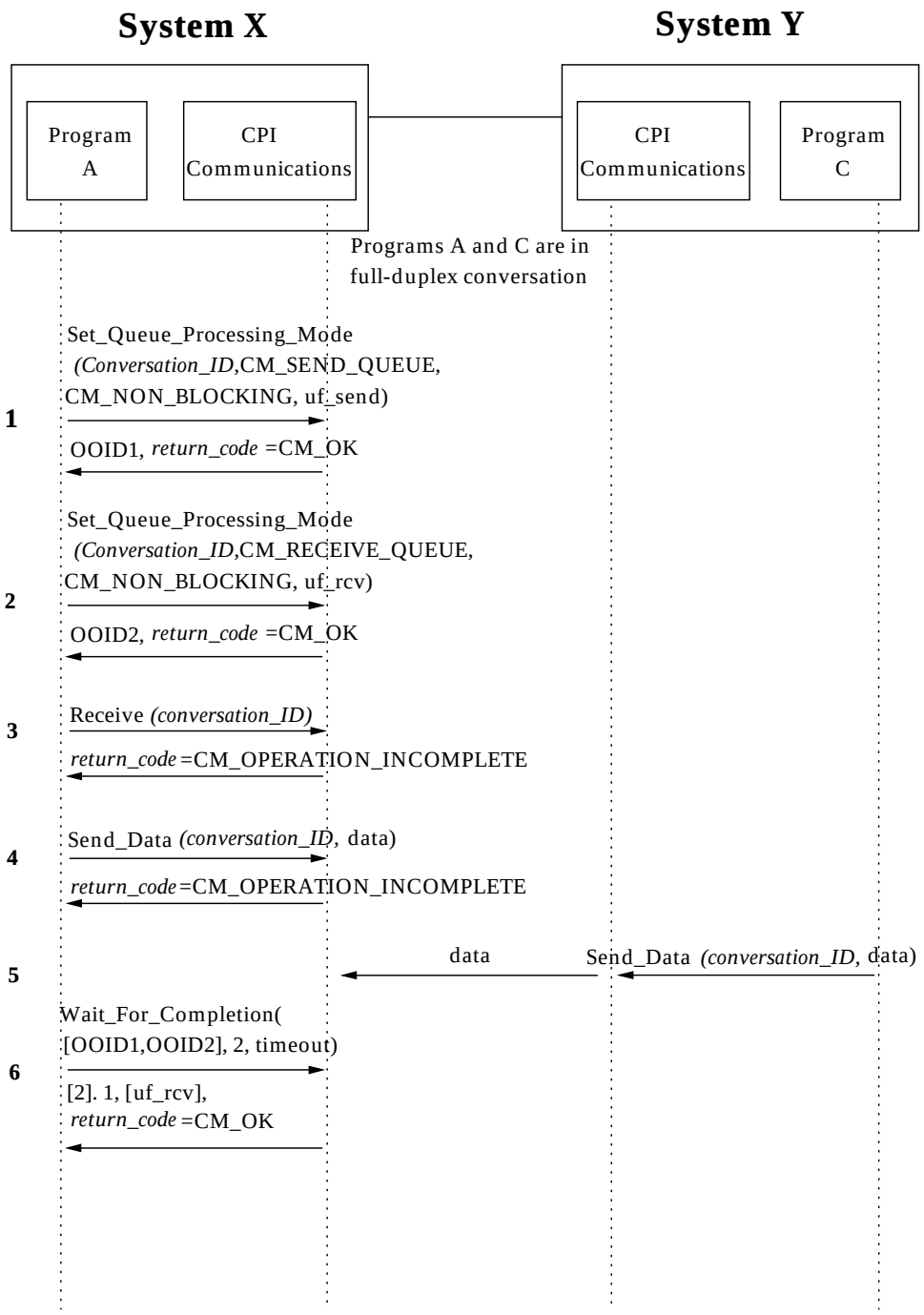
4.3.12 Using Queue-level Non-blocking

Figure 4-13 on page 93 shows an example of a program that uses queue-level non-blocking.

The steps shown in Figure 4-13 on page 93 are:

Step	Description
1	In a full-duplex conversation, the state of the conversation at Program A and Program C is **Send-Receive** state. Program A issues Set_Queue_Processing_Mode to set the processing mode for its Send queue to CM_NON_BLOCKING. It also specifies a user field, uf_send, as a pointer to the parameters on the Send_Data call. When the Set_Queue_Processing_Mode call completes successfully, Program A receives an OOID, **OOID1**, that is unique to the Send queue.
2	Program A also issues Set_Queue_Processing_Mode to set the processing mode for its Receive queue to CM_NON_BLOCKING. This time it specifies a user field, uf_rcv, as a pointer to the parameters on the Receive call. When the Set_Queue_Processing_Mode call completes successfully, Program A receives an OOID, **OOID2**, that is unique to the Receive queue.
3	Program A issues a Receive call. Because no incoming data is ready to be received, the call is suspended and returns CM_OPERATION_INCOMPLETE.
4	Program A issues a Send_Data call, which also returns CM_OPERATION_INCOMPLETE because of transmission buffer shortage.
5	Program C sends data to Program A, which will satisfy the outstanding Receive call.
6	Program A issues Wait_For_Completion to wait for both outstanding operations. It does so by specifying **OOID1** and **OOID2** in the *OOID_list*. To indicate that the Receive call has completed, the Wait_For_Completion call returns an index value 2 and uf_rcv, which are associated with the Receive call. Program A can now use the returned user field, uf_rcv, to examine the return code of the Receive call.

Figure 4-13 Using Queue-level Non-blocking



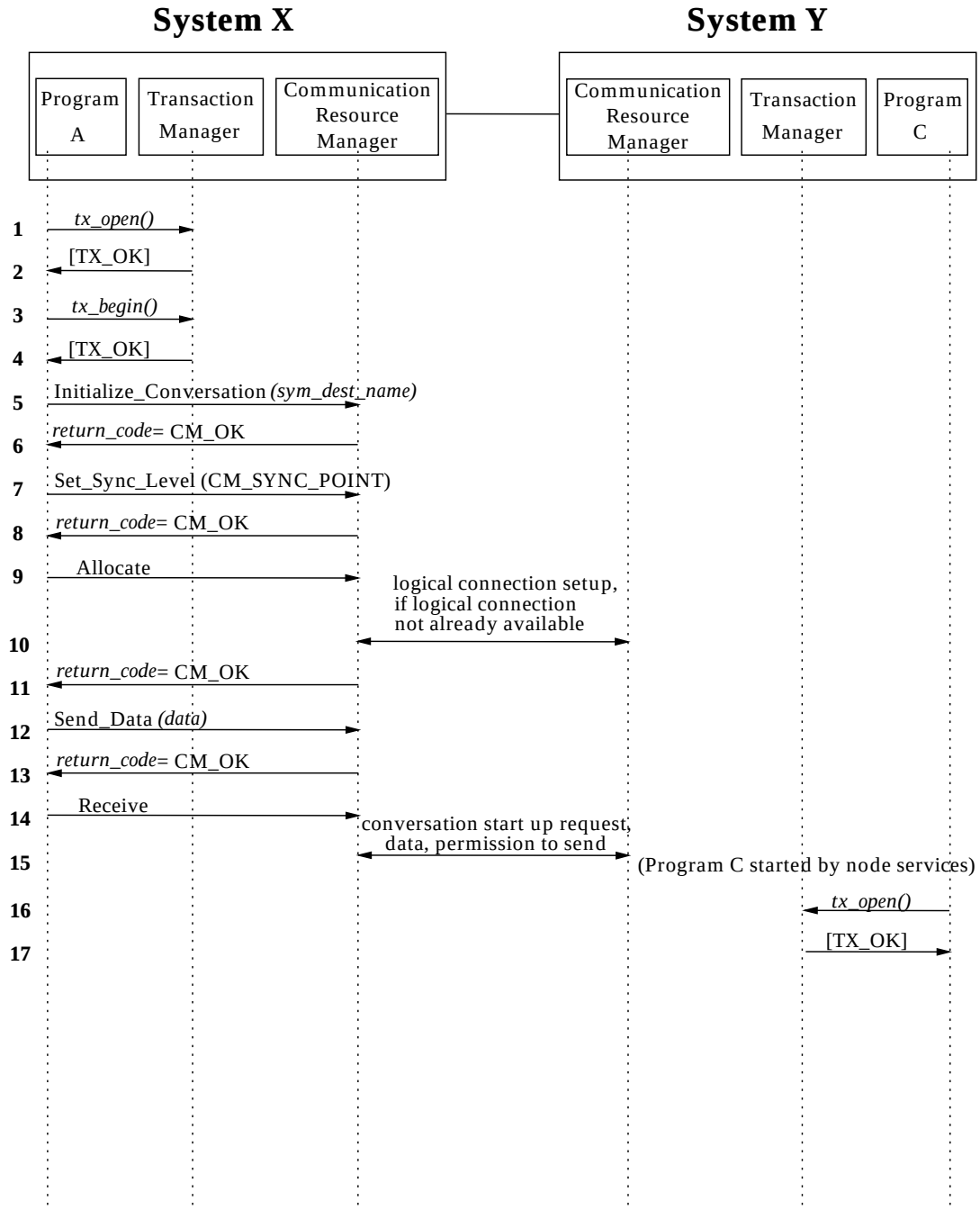
4.3.13 Sending Program Issues a Commit

Figure 4-14 on page 95 shows a program sending data on a protected half-duplex conversation and invoking the TX (Transaction Demarcation) interface to issue a commit instruction. A protected conversation is one in which the *sync_level* has been set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM. 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 4-14 on page 95 are:

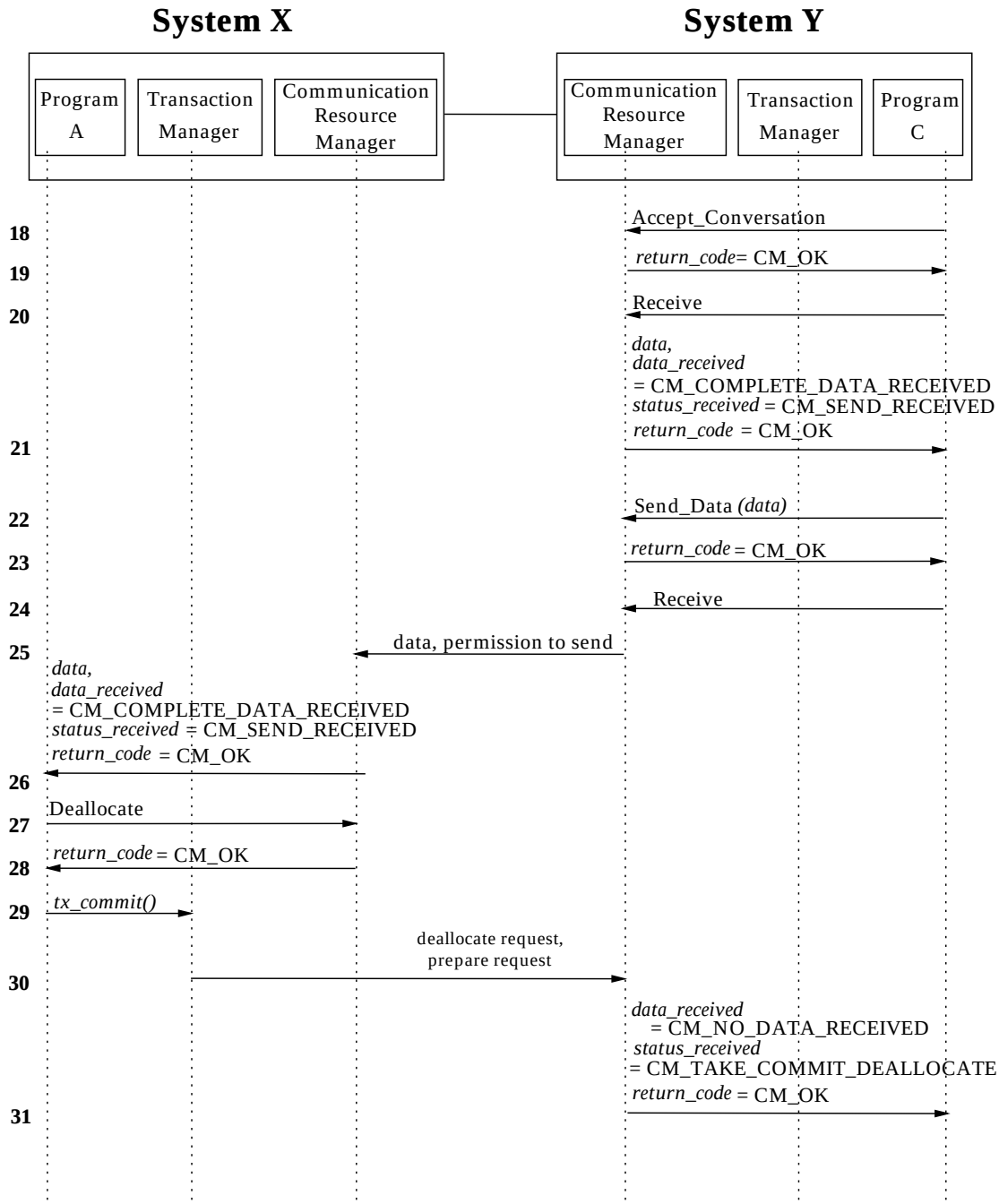
Step	Description
1	Program A opens all resource managers linked with the program. These are the CPI-C communication resource manager and possibly some resource managers of database systems.
2	The TX function *tx_open*( ) returns [TX_OK], indicating that all the resource managers have been opened.
3	Program A issues a *tx_begin*( ) call in order to start a transaction.
4	The function *tx_begin*( ) returns [TX_OK], indicating that Program A is now in transaction mode and that all local resource managers have been included in the transaction.
5 and 6	To communicate with its partner program, Program A must first establish a conversation.
7 and 8	Program A uses the Set_Sync_Level call to request that the conversation be part of the transaction.
9 to 11 inclusive	Program A issues the Allocate call to start the conversation.
12 and 13	Program A sends data.
14	Program A issues a Receive call indicating that it is now ready to receive data from Program C.
15	The CPI-C communication resource manager sends the conversation startup request, data and the permission to send. When using the OSI TP protocol, these are TP-BEGIN-DIALOGUE-RI, C-BEGIN-RI, UASE-RI and TP-GRANT-CONTROL-RI. Program C is now started by node services.
16	Program C opens all resource managers linked with the program. These are the CPI-C communication resource manager and possibly some resource managers of database systems. The function *tx_open*( ) must be issued before the first CPI-C call.
17	The function *tx_open*( ) returns [TX_OK], indicating that all the resource managers have been opened.

Figure 4-14 Establishing a Protected Conversation and Issuing a Successful Commit



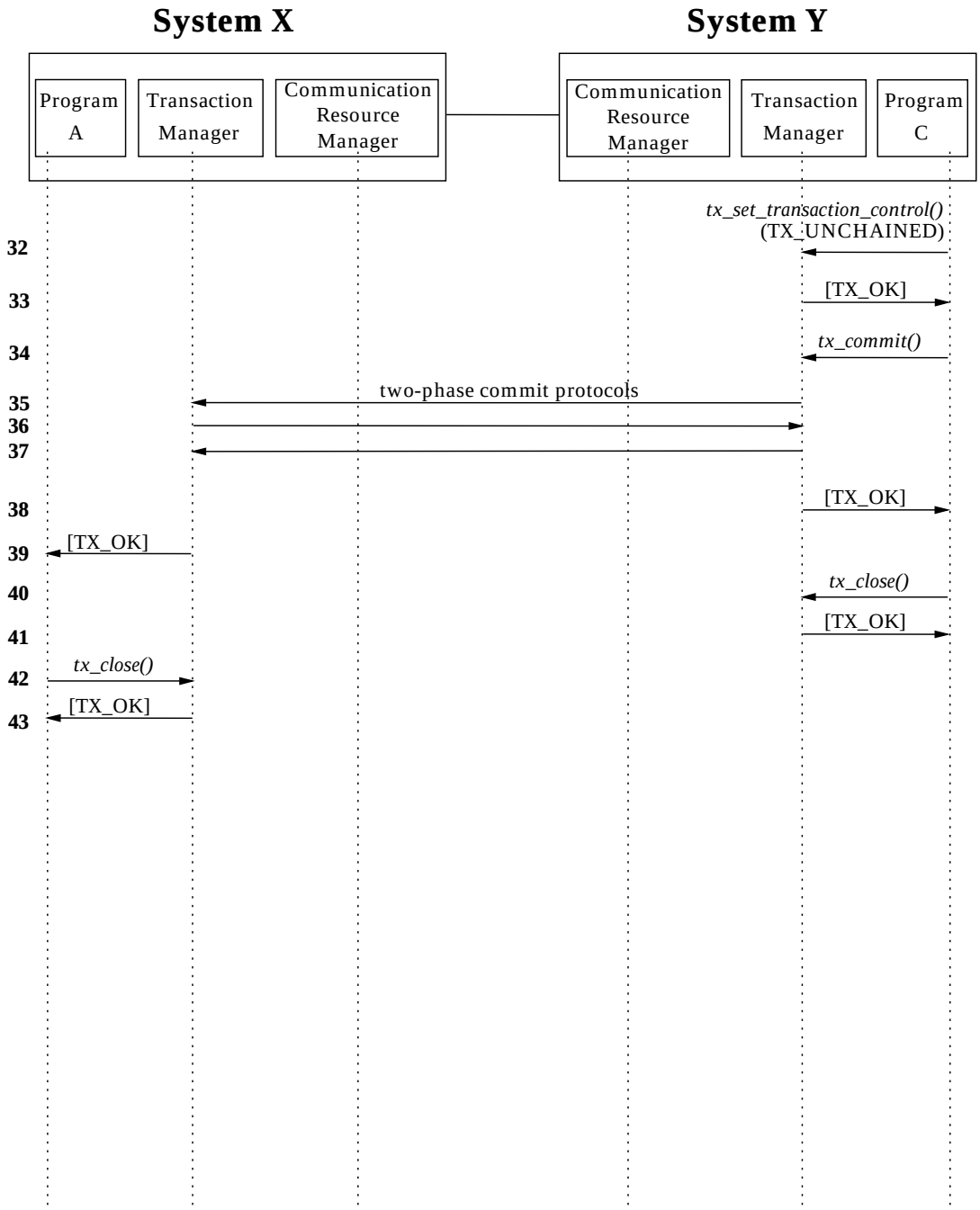
Step	Description
18 and 19	Program C issues an Accept_Conversation call. While processing this call, CPI Communications implicitly issues a *tx_begin*( ) call and a *tx_set_transaction_control*( ) call with control set to TX_CHAINED. After this call the program may extract some CPI-C and TX characteristics to obtain the following information: Extract_Sync_Level — *sync_level*=CM_SYNC_POINT Extract_Transaction_Control — *transaction_control*=CM_CHAINED_TRANSACTIONS *tx_info*() — return code = 1 (caller is in transaction mode), *transaction_control*=TX_CHAINED
20 and 21	Program C receives the data sent by Program A. With the same call (or in the next Receive call) CPI-C indicates that Program C is now in **Send** state. Program C may now issue any database calls to the local database.
22 and 23	Program C sends data.
24	Program C issues a Receive call to flush the send buffer and wait for the take-commit notification.
25	The CPI-C communication resource manager sends the data and the permission to send. When using the OSI TP protocol, these are TP-BEGIN-DIALOGUE-RC, C-BEGIN-RC, UASE-RI and TP-GRANT-CONTROL-RI.
26	Because of incoming data, Program A's Receive call executes successfully and it receives the data sent by Program C. With the same call (or in the next Receive call), CPI-C indicates that Program A is now in Send state. Program A may now issue any database calls to update the local database.
27 and 28	Program A issues a Deallocate call to end the conversation. The *deallocate_type* characteristic has its initial value set to CM_DEALLOCATE_SYNC_LEVEL. Since the *sync_level* characteristic is set to CM_SYNC_POINT, Program A's end of the conversation is now in **Defer-Deallocate** state.
29	Program A issues a *tx_commit*( ) call to make all of the updates permanent and to advance all of the protected resources to a synchronization point.
30	The transaction manager informs the CPI-C communication resource manager. The CPI-C communication resource manager sends the Deallocate and the Prepare request. When using the OSI TP protocol, these are TP-DEFER-RI(end-dialogue) and C-PREPARE-RI.
31	Program C's Receive call executes successfully and it receives the take-commit notification as a CM_TAKE_COMMIT_DEALLOCATE value in the *status_received* parameter of its Receive call. Program C's end of the conversation is now in **Sync-Point-Deallocate** state.

Figure 4-14 on page 95 continued.



Step	Description
32 and 33	Because the *status_received* value is CM_TAKE_COMMIT_DEALLOCATE, Program C knows that it should issue a *tx_commit*( ) call to confirm the sync point. This *tx_commit*( ) call ends the conversation. As a consequence, Program C sets the TX *transaction_control* characteristic to TX_UNCHAINED. If this call is omitted, the following *tx_commit*( ) call starts a new transaction. This transaction is only local and may only be used to co-ordinate updates on different local databases.
34	Program C responds to the take-commit notification by issuing the *tx_commit*( ) call. The CM_TAKE_COMMIT_DEALLOCATE value means that, if this commit is successful, Program C's end of the conversation is deallocated (put in **Reset** state).
35 to 37 inclusive	Two phase commit protocols are exchanged between the two transaction managers. When using the OSI TP protocol, these are C-READY-RI, C-COMMIT-RI, C-COMMIT-RC.
38 and 39	Both Program A and Program C receive return codes that indicate successful completion of the commit operation. Note that both Programs A and C may have issued any database calls between step 4 and step 34. The return code [TX_OK] from *tx_commit*( ) indicates that all these database updates on both systems have been successful. The conversation between Program A and Program C is now deallocated. Both ends of the conversations are in **Reset** state. **Note:** If the *tx_commit*( ) is unsuccessful and responses indicating backout are received by Programs A and C, for example the return code is [TX_ROLLBACK], the conversation is not deallocated. The conversation states for Program A and C are reset to their values at the time of the last sync point. Because in this example there was no prior sync point, Program A's end of the conversation goes to **Send** state, and Program C's end of the conversation goes to **Receive** state. The program can retrieve the current conversation state by using the CPI Communications Extract_Conversation_State call.
40 to 43 inclusive	Both Program A and Program C issue a *tx_close*( ) call to close all resource managers linked with the program.

Figure 4-14 on page 95 continued.



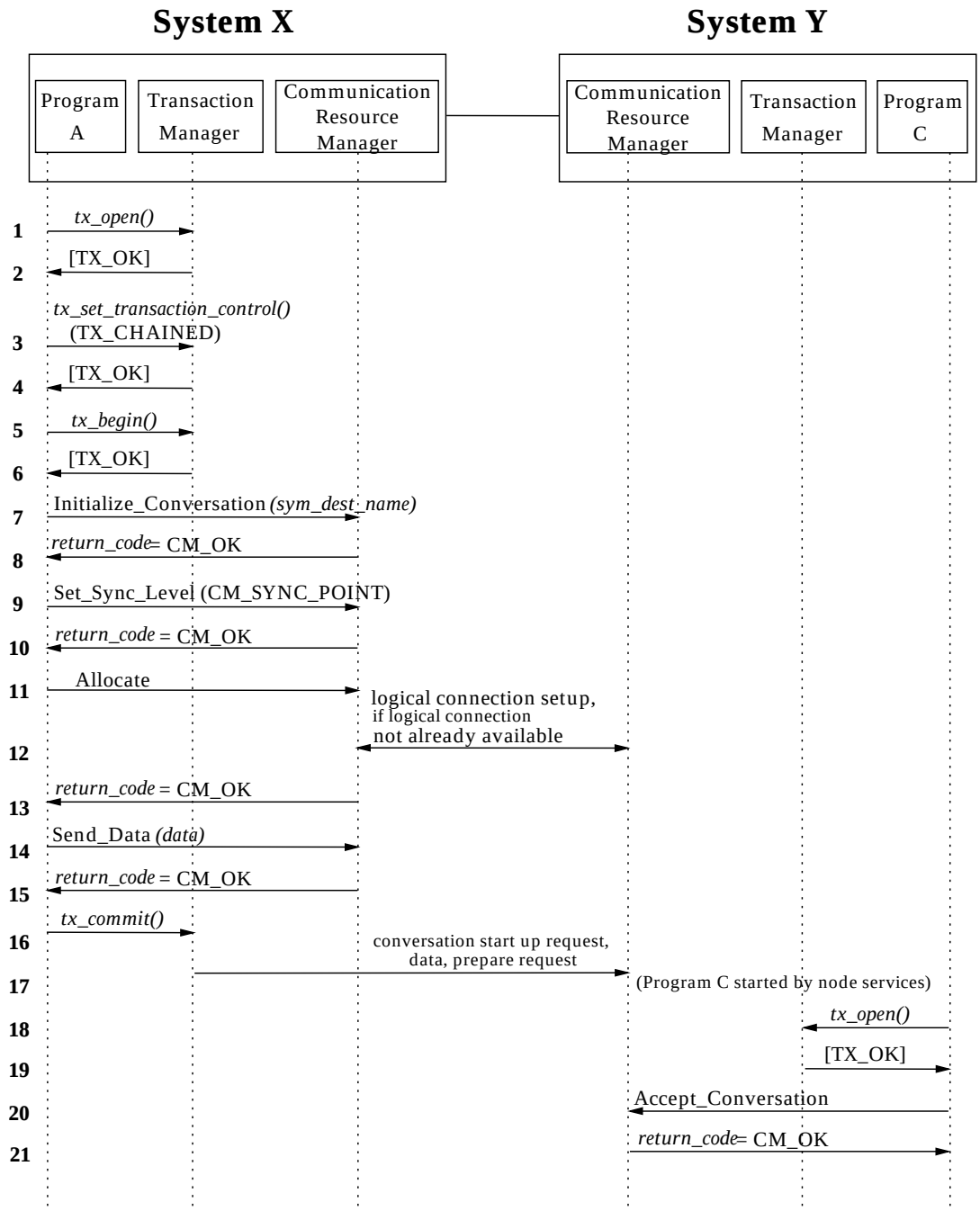
4.3.14 Two Chained Transactions

Figure 4-15 on page 101 shows a conversation between two programs with two chained transactions on a half-duplex conversation. This means that there are two sync points in one conversation; the second transaction starts immediately after the commit of the first transaction.

The steps shown in Figure 4-15 on page 101 are:

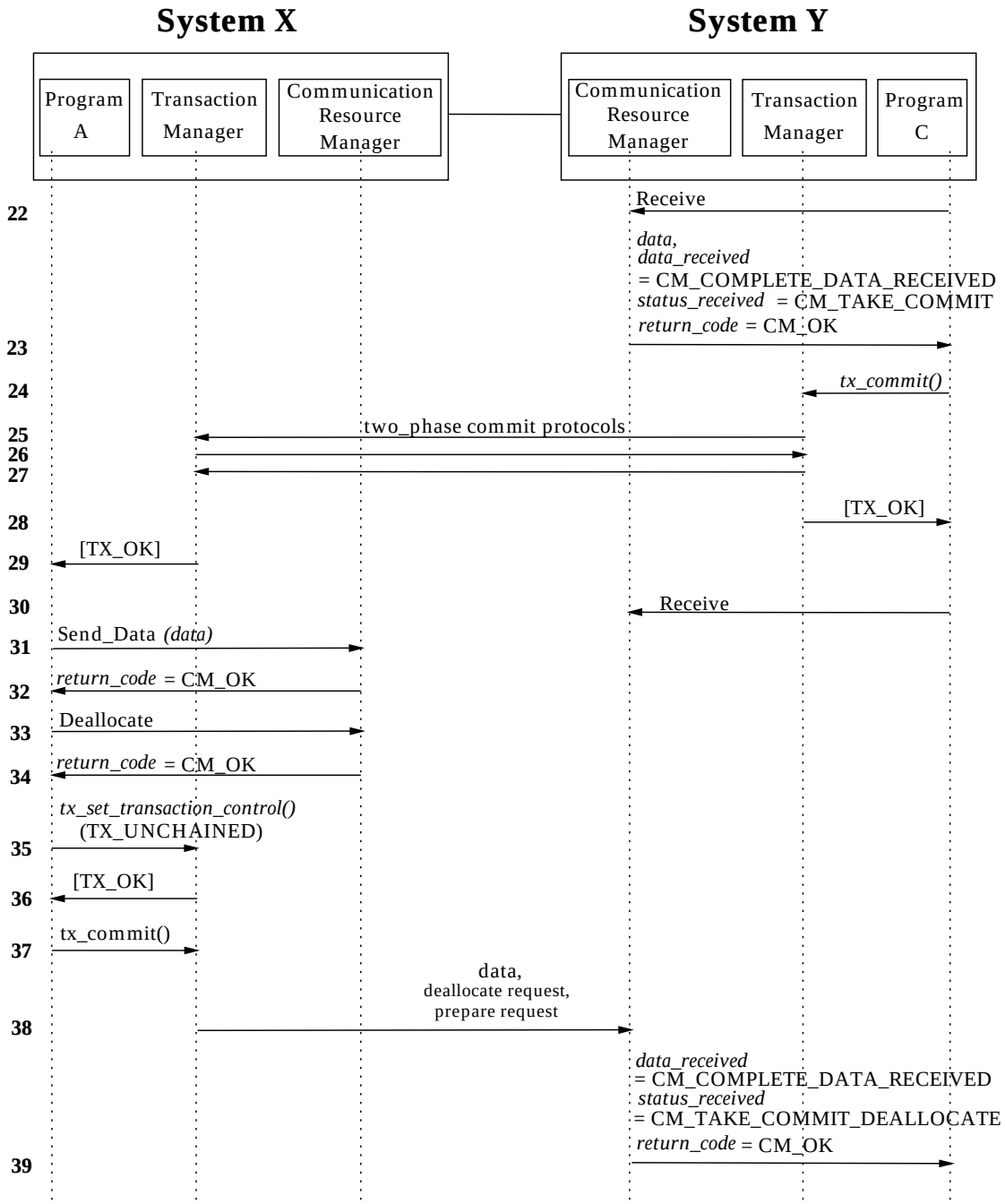
Step	Description
1 and 2	Program A opens all resource managers linked with the program.
3 and 4	Program A sets the TX transaction control characteristic to TX_CHAINED. This means that a *tx_commit*( ) call should start a new transaction before returning to the caller. If this call is omitted, the conversation would be deallocated by a CM_RESOURCE_FAILURE_RETRY return code after step 29.
5 and 6	Program A issues a *tx_begin*( ) call in order to start a transaction.
7 to 13 inclusive	To communicate with its partner program, Program A must first establish a conversation. It uses the Set_Sync_Level call in step 9 to request that the conversation be protected. Note that the CPI-C *transaction_control* characteristic has its standard value CM_CHAINED_TRANSACTIONS.
14 and 15	Program A sends data. Before or after this call, Program A may issue any database calls updating the local database.
16	Program A issues a *tx_commit*( ) call to make all of the updates permanent and to advance all of the protected resources to a synchronization point.
17	The CPI-C communication resource manager sends the conversation start up request, the data and the prepare request. When using the OSI TP protocol, these are TP-BEGIN-DIALOGUE-RI, C-BEGIN-RI, UASE-RI and C-PREPARE-RI. Program C is now started by node services.
18 and 19	Program C opens all resource managers linked with the program.
20 and 21	Program C issues an Accept_Conversation call. While processing this call, CPI Communications implicitly issues a *tx_begin*( ) call and a *tx_set_transaction_control*( ) call with control set to TX_CHAINED.

Figure 4-15 Two Chained Transactions



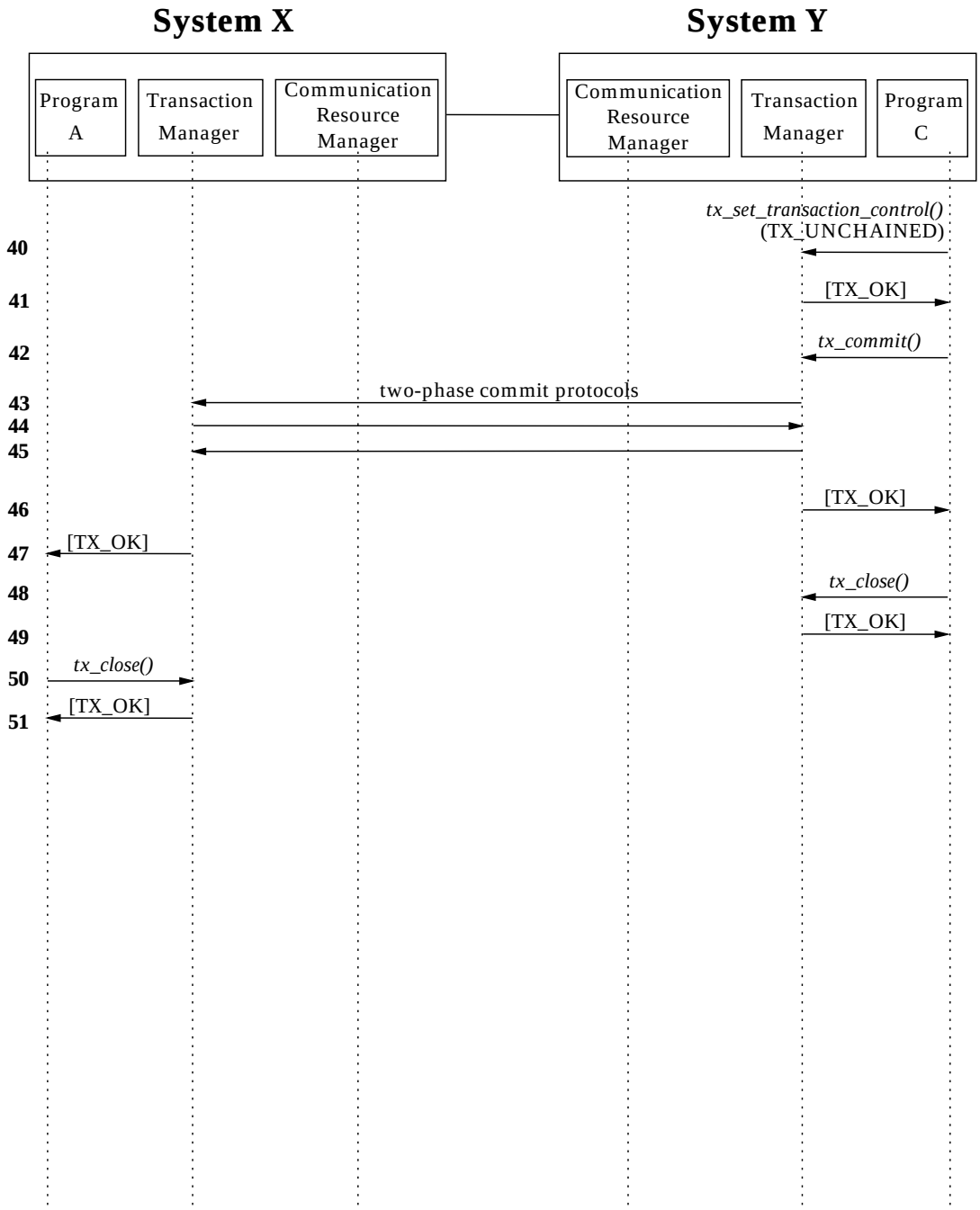
Step	Description
22 and 23	Program C receives the data sent by Program A. With the same call or in the next Receive call) CPI-C indicates the take-commit notification in the *status_received* parameter. Program C's end of the conversation is now in **Syncpoint** state. Program C may now issue any database calls updating the local database.
24	Program C responds to the take-commit notification by issuing the *tx_commit*( ) call.
25 to 27 inclusive	Two phase commit protocols are exchanged between the two transaction managers. When using the OSI TP protocol, these are C-READY-RI, C-COMMIT+C-BEGIN-RI and C-COMMIT-RC.
28 and 29	Both Program A and Program C receive return codes that indicate successful completion of the commit operation. Program A's end of the conversation is now in **Send** state, and Program C's end of the conversation is now in **Receive** state. Because the TX *transaction_control* characteristic is set to TX_CHAINED on both sides, both programs are again in transaction mode. Because the CPI-C *transaction_control* characteristic is set to CM_CHAINED_TRANSACTIONS, the conversation is automatically included in the new transaction.
30	Program C issues a Receive call.
31 and 32	Program A sends data. Before or after this call, Program A may issue any database calls updating the local database.
33 and 34	Program A issues a Deallocate call to end the conversation.
35 and 36	Program A sets the TX *transaction_control* characteristic to TX_UNCHAINED. This call is needed because the following *tx_commit*( ) call must not start a new transaction. If this call is omitted, the transaction started by the following *tx_commit*( ) call is only local and may be used only to co-ordinate updates on different local data bases.
37	Program A issues a *tx_commit*( ) call to make all of the updates permanent and to advance all of the protected resources to a second synchronization point.
38	The CPI-C communication resource manager sends the data, deallocate request and prepare request.
39	Program C's Receive call executes successfully and it receives the data sent by program A. With the same call (or in the next Receive call), CPI-C indicates the take-commit notification as a CM_TAKE_COMMIT_DEALLOCATE.

Figure 4-15 on page 101 continued.



Step	Description
40 and 41	Program C sets the TX *transaction_control* characteristic to TX_UNCHAINED. If this call is omitted, the following *tx_commit*( ) starts a new transaction. This transaction is only local and may be used only to co-ordinate updates on different local data bases.
42	Program C responds to the take-commit notification by issuing the *tx_commit*( ) call.
43 to 45 inclusive	Two phase commit protocols are exchanged between the two transaction managers.
46 and 47	Both Program A and Program C receive the return code [TX_OK] indicating the successful completion of the commit operation. **Note:** If the *tx_commit*( ) is unsuccessful the database updates are reset to the last sync point. This means that all the updates between steps 29 and 42 are reset.
48 to 51 inclusive	Both Program A and Program C issue a *tx_close*( ) call to close all resource managers linked with the program.

Figure 4-15 on page 101 continued.



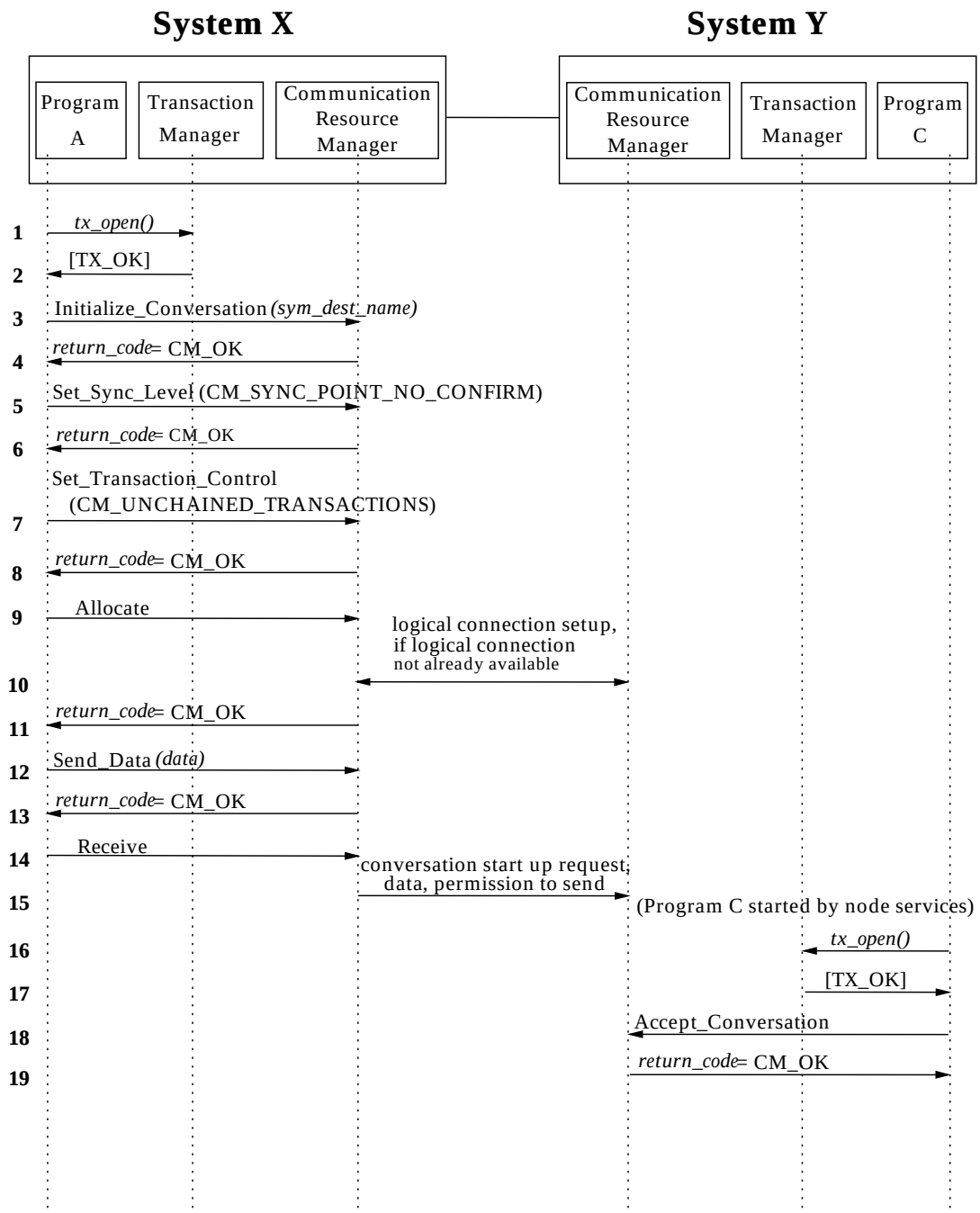
4.3.15 Unchained Transactions

Figure 4-16 on page 107 shows a conversation between two programs with unchained transactions on a half-duplex conversation. This means that parts of the conversation are protected while other parts are not.

The steps shown in Figure 4-16 on page 107 are:

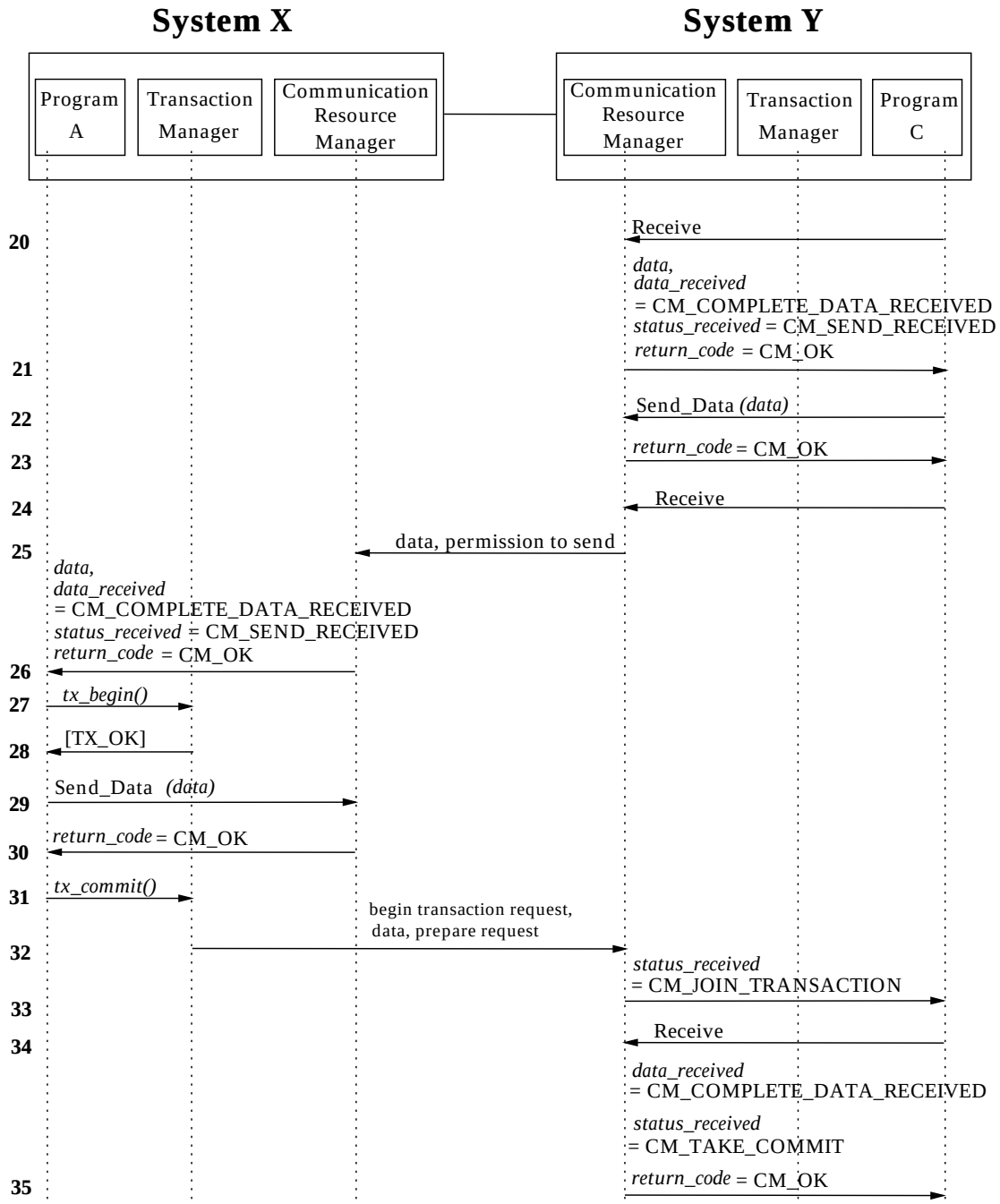
Step	Description
1 and 2	Program A opens all resource managers linked with the program. The TX *transaction_control* characteristic has its initial value set to TX_UNCHAINED.
3 and 4	To communicate with its partner program, Program A must first establish a conversation.
5 and 6	Program A sets the *sync_level* characteristic to CM_SYNC_POINT_NO_CONFIRM. This means that the programs can perform sync point processing on this conversation, but they cannot perform confirmation processing.
7 and 8	Program A sets the CPI-C *transaction_control* characteristic to CM_UNCHAINED_TRANSACTIONS.
9 to 11 inclusive	Program A issues an Allocate call. Because of CM_UNCHAINED_TRANSACTIONS and the missing *tx_begin*( ), this call does not ask the partner to join the transaction. The actual conversation is not included in a transaction.
12 and 13	Program A sends data.
14	Program A issues a Receive call indicating that it is now ready to receive data from Program C.
15	The CPI-C communication resource manager sends the conversation startup request, data and the permission to send. When using the OSI TP protocol, these are TP-BEGIN-DIALOGUE-RI, UASE-RI and TP-GRANT-CONTROL-RI. Program C is now started by node services.
16 and 17	Program C opens all resource managers linked with the program.
18 and 19	Program C issues an Accept_Conversation call. After this call the program may extract some CPI-C and TX characteristics to obtain the following information: Extract_Sync_Level — *sync_level*=CM_SYNC_POINT_NO_CONFIRM Extract_Transaction_Control — *transaction_control*=CM_UNCHAINED_TRANSACTIONS *tx_info*() — return code = 0 (caller is not in transaction mode).

Figure 4-16 Unchained Transactions



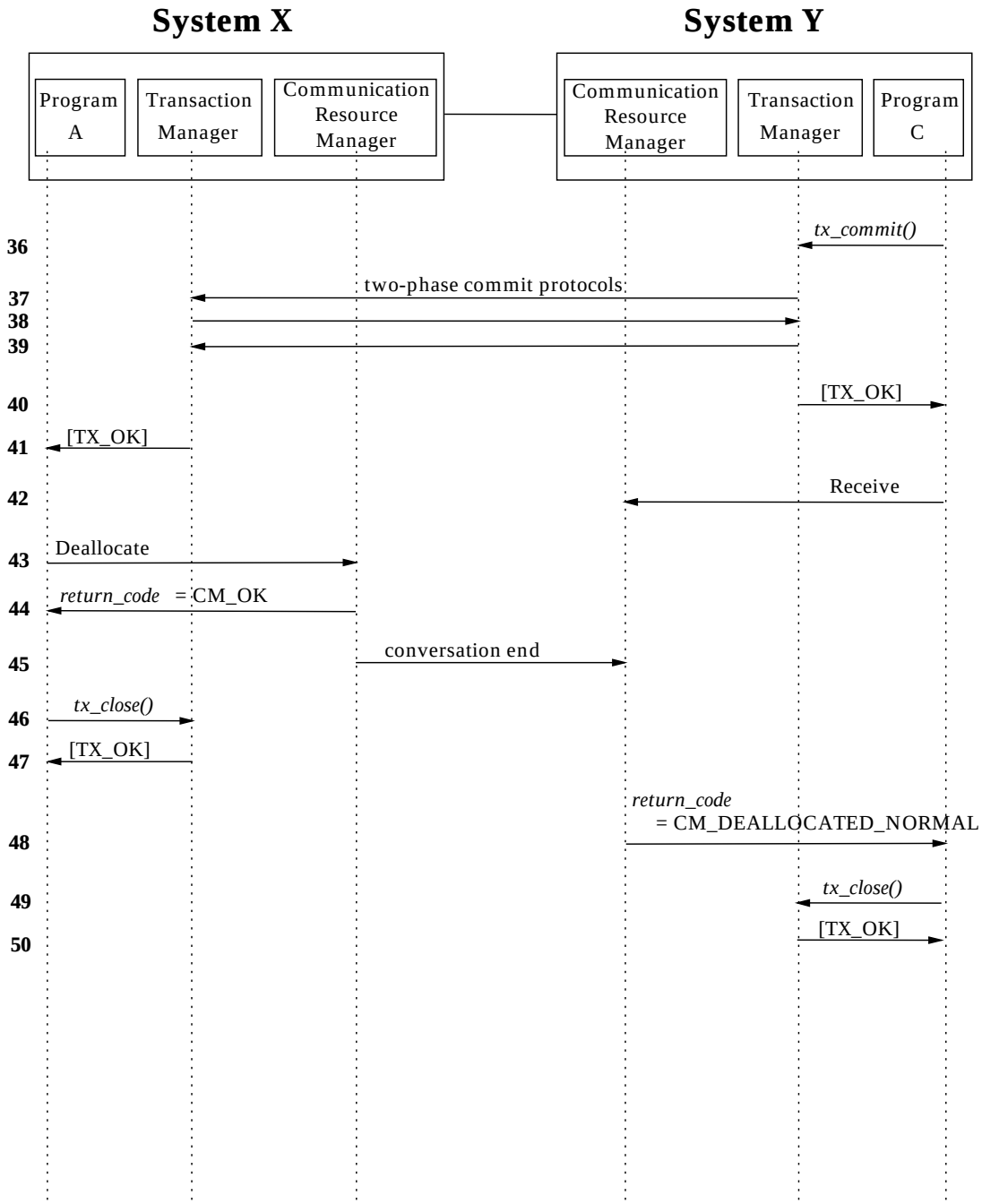
Step	Description
20 and 21	Program C receives the data sent by Program A. With the same call (or in the next Receive call), CPI-C indicates in the *status_received* parameter that the conversation is now in **Send** state.
22 and 23	Program C sends data.
24	Program C issues a Receive call.
25	The CPI-C communication resource manager sends the data and the permission to send. When using the OSI TP protocol, these are TP-BEGIN-DIALOGUE-RC, UASE-RI and TP-GRANT-CONTROL-RI.
26	Because of incoming data Program A's Receive call executes successfully and it receives the data sent by Program C. With the same call (or in the next Receive call) CPI-C indicates that the conversation is now in **Send** state.
27 and 28	Program A issues a *tx_begin*( ) call in order to start a transaction. Neither program should issue any co-ordinated database updates before this call. In case of a failure between step 1 and step 21, databases may become inconsistent.
29 and 30	Program A sends data. Because of the preceding *tx_begin*( ) call and because the *begin_transaction* characteristic has its initial value CM_BEGIN_IMPLICIT, this call asks Program C to join the transaction. So the actual conversation is included in the current transaction.
31	Program A issues a *tx_commit*( ) call to make all of the updates permanent and to advance all of the protected resources to a synchronization point.
32	The CPI-C communication resource manager sends the begin transaction request, data and prepare request. When using the OSI TP protocol these are C-BEGIN-RI, UASE-RI and C-PREPARE-RI.
33	Because of the incoming begin transaction request, CPI Communications implicitly issues a *tx_begin*( ) call to join the transaction. Program C's Receive call executes successfully and it receives a *status_received* value of CM_JOIN_TRANSACTION. A *tx_info*( ) call issued after the Receive call would return the following values: *tx_info*() return_code = 1 (caller is in transaction) *transaction_control*=TX_UNCHAINED
34 and 35	Program C issues a Receive call to receive the data sent by Program A. With the same call (or in the next Receive call), CPI-C indicates the *take-commit* notification in the *status_received* parameter.

Figure 4-16 on page 107 continued.



Step	Description
36	Program C responds to the take-commit notification by issuing the *tx_commit*( ) call.
37 to 39 inclusive	Two phase commit protocols are exchanged between the two transaction managers. When using the OSI TP protocol these are C-BEGIN-RC+C-READY-RI, C-COMMIT-RI and C-COMMIT-RC.
40 and 41	Both programs receive the return code [TX_OK] indicating the successful completion of the commit operation. Program A's end of the conversation is now in **Send** state, and Program C's end of the conversation is now in **Receive** state. Because the TX *transaction_control* characteristic is set to TX_UNCHAINED on both sides, neither program is in transaction mode. Because the CPI-C *transaction_control* characteristic is set to CM_UNCHAINED_TRANSACTIONS, the conversation is not included in a transaction. **Note:** If the *tx_commit*( ) call is unsuccessful, all the database updates between steps 21 and 36 are reset to the last sync point. Database updates before step 21 are not reset.
42	Program C issues a Receive call.
43 and 44	Program A issues a Deallocate call to end the conversation. Because Program A is not in transaction mode and the *sync_level* characteristic is set to CM_SYNC_POINT_NO_CONFIRM, the conversation deallocates without a new sync point and without confirmation. In case of *sync_level* CM_SYNC_POINT, it would be a deallocate with confirmation.
45	The CPI-C communication resource manager sends the conversation end. When using the OSI TP protocol this is a TP-END-DIALOGUE-RI.
46 and 47	Program A issues a *tx_close*( ) call.
48	Program C's Receive call executes successfully and it receives the return_code CM_DEALLOCATED_NORMAL.
49 and 50	Program C issues a *tx_close*( ) call.

Figure 4-16 on page 107 continued.



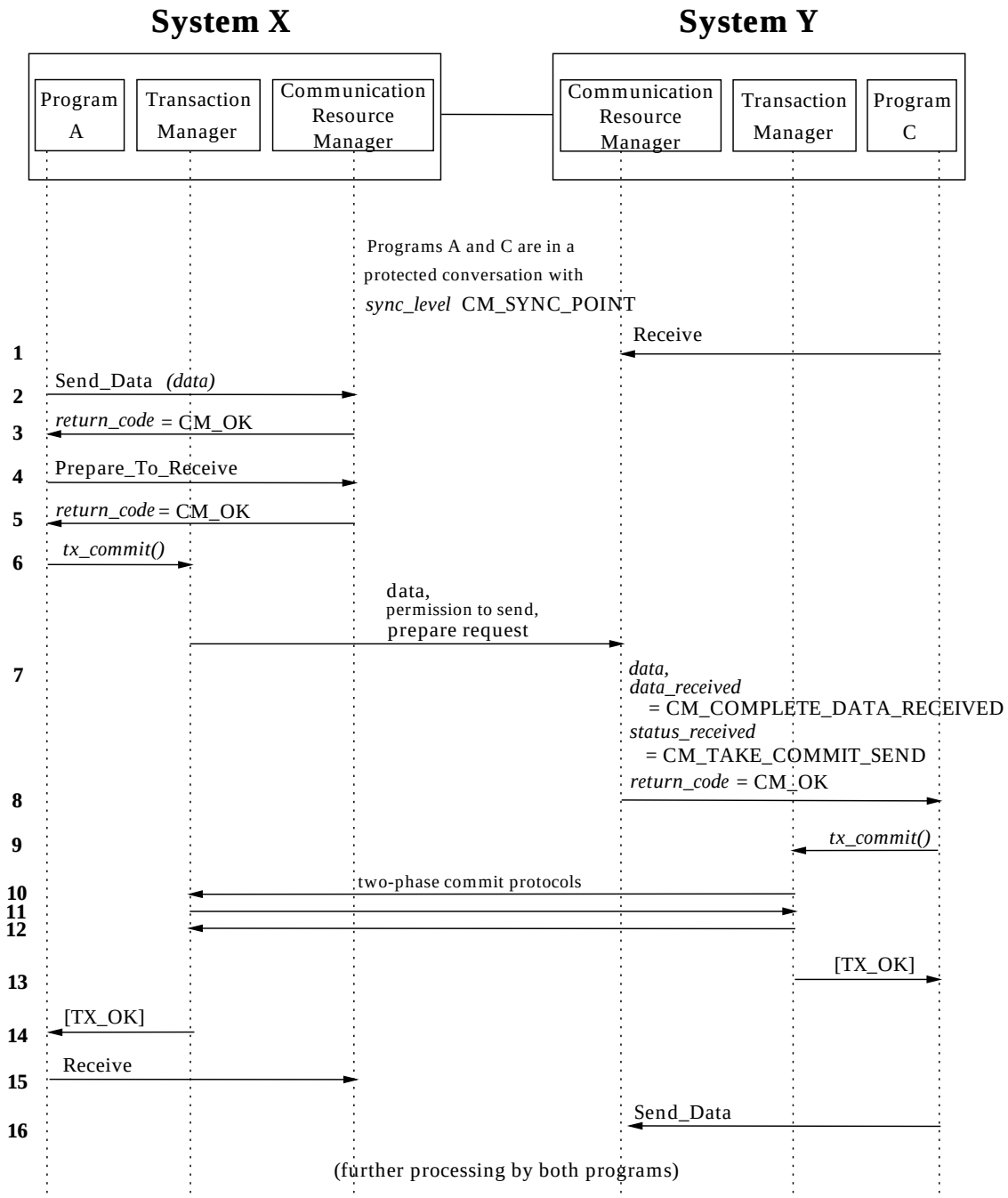
4.3.16 Successful Commit with Conversation State Change

Figure 4-17 on page 113 shows a successful commit with a conversation state change on a half-duplex conversation.

The steps shown in Figure 4-17 on page 113 are:

Step	Description
1	Programs A and C are in a protected conversation with *sync_level* CM_SYNC_POINT. Program C's end of the conversation is in **Receive** state. It issues a Receive call.
2 and 3	Program A's end of the conversation is in **Send** state. It sends data.
4 and 5	Program A wants its side of the conversation to be changed from **Send** to **Receive** state after it issues its next commit call. To do this, Program A uses the Prepare_To_Receive call. It is assumed that the *prepare_to_receive_type* characteristic is on its initial value CM_PREPARE_TO_RECEIVE_SYNC_LEVEL. After the call, Program A's side of the conversation is in **Defer-Receive** state until Program A issues a commit call.
6	Program A issues a *tx_commit*( ) call in **Defer-Receive** state. If the call completes successfully, Program A's end of the conversation is placed in **Receive** state.
7	The CPI-C communication resource manager sends the data, the permission to send and the prepare request. When using the OSI TP protocol, these are UASE-RI, TP-DEFER-RI(grant-control) and C-PREPARE-RI.
8	Because Program A was in **Defer-Receive** state when it issued the commit call, the CPI-C communication resource manager returns the take-commit notification to Program C as a CM_TAKE_COMMIT_SEND value in the *status_received* parameter. This value means that Program C's end of the conversation is now in **Syncpoint-Send** state. It is placed in **Send** state if Program C completes a commit call successfully.
9	Program C responds to the take-commit notification with a *tx_commit*( ) call.
10 to 12 inclusive	Two phase commit protocols are exchanged between the two transaction managers.
13 and 14	Both *tx_commit*( ) calls end successfully. **Note:** If the *tx_commit*( ) is unsuccessful. and Programs A and C receive responses indicating backout, for example the return code [TX_ROLLBACK], 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.
15	Program A's end of the conversation is now in **Receive** state and it issues a Receive call.
16	Program C's end of the conversation is now in **Send** state and it issues a Send_Data call.

Figure 4-17 Successful Commit with Conversation State Change



Chapter 5

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:

SYNOPSIS

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 Section 5.1 on page 116 for specifics.

DESCRIPTION

A description of the call and the parameters that are required. 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 Section 3.13 on page 49 for more information on conversation states.

APPLICATION USAGE

Additional information that applies to the call.

SEE ALSO

Where to find additional information related to the call.

5.1 Call Syntax

CPI Communications calls can be made from application programs written in the following programming languages:

- C
- COBOL.

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

This specification 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:

C

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

COBOL

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

5.2 Programming Language Considerations

This section describes the programming language considerations a programmer should keep in mind when writing and running a program that uses CPI Communications. Sample pseudonym files are provided in Appendix E for the C and COBOL programming languages. Customised 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.

5.2.1 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.
- To enable asynchronous updates of program variables, the return parameters on a non-blocking call must be declared using the *volatile* qualifier as defined in ANSI C.

5.2.2 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-5, not zoned decimal.

5.3 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 specification. 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 305 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*.
- Section 5.1 on page 116 shows the syntax for the programming language being used.
- Appendix A provides a complete description of all variables used in the specification 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 A-3 on page 341.
- Table A-1 on page 330 in Appendix A 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. CM_OK means that the call completed successfully.

5.4 Locations of Key Topics

Table 5-1 on page 120. provides a summary list of program calls, in pseudonym sequence. Against each pseudonym is shown the actual call name and a brief description.

Key-topic discussions and where they occur are:

- Section 1.3 on page 3 describes the naming conventions used throughout the specification.
- Section 4.3.1 on page 71 discusses program control over data transmission.
- The **APPLICATION USAGE** section of *Request_To_Send (CMRTS)* on page 227 discusses how a conversation enters **Receive** state.
- The **APPLICATION USAGE** section of *Send_Data (CMSEND)* on page 230 describes the use of logical records and LL fields on basic conversations.

Table 5-1 Summary List of Calls and their Descriptions

Pseudonym	Call	Description
Accept_Conversation	CMACCP	Used by a program to accept an incoming conversation.
Accept_Incoming	CMACCI	Used by a program to accept an incoming conversation previously initialized with the Initialize_For_Incoming call.
Allocate	CMALLC	Used by a program to establish a conversation.
Cancel_Conversation	CMCANC	Used by a program to end a conversation immediately.
Confirm	CMCFM	Used by a program to send a confirmation request to its partner.
Confirmed	CMCFMD	Used by a program to send a confirmation reply to its partner.
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.
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.
Deallocate	CMDEAL	Used by a program to end a conversation.
Deferred_Deallocate	CMDFDE	Used by a program to end a conversation following successful completion of the current transaction.
Extract_AE_Qualifier	CMEAEQ	Used by a program to view the current *AE_qualifier* conversation characteristic.
Extract_AP_Title	CMEAPT	Used by a program to view the current *AP_title* conversation characteristic.
Extract_Application_Context_Name	CMEACN	Used by a program to view the current *application_context_name* conversation characteristic.
Extract_Conversation_State	CMECS	Used by a program to view the current state of a conversation.
Extract_Conversation_Type	CMECT	Used by a program to view the current *conversation_type* conversation characteristic.

Pseudonym	Call	Description
Extract_Initialization_Data	CMEID	Used by a program to extract the current *initialization_data* conversation characteristic.
Extract_Maximum_Buffer_Size	CMEMBS	Used by a program to extract the maximum buffer size supported by the system.
Extract_Mode_Name	CMEMN	Used by a program to view the current *mode_name* conversation characteristic.
Extract_Partner_LU_Name	CMEPLN	Used by a program to view the current *partner_LU_name* conversation characteristic.
Extract_Secondary_Information	CMESI	Used by a program to extract secondary information associated with the return code for a given call.
Extract_Security_User_ID	CMESUI	Used by a program to view the current *security_user_ID* conversation characteristic.
Extract_Send_Receive_Mode	CMESRM	Used by a program to view the current *send_receive_mode* conversation characteristic.
Extract_Sync_Level	CMESL	Used by a program to view the current *sync_level* conversation characteristic.
Extract_TP_Name	CMETPN	Used by a program to determine the *TP_name* characteristic's value for a given conversation.
Extract_Transaction_Control	CMETC	Used by a program to extract the *transaction_control* characteristic's value for a given conversation.
Flush	CMFLUS	Used by a program to flush the local CRM's send buffer.
Include_Partner_In_Transaction	CMINCL	Used by a program to include a partner program in a transaction.
Initialize_Conversation	CMINIT	Used by a program to initialize the conversation characteristics for an outgoing conversation.
Initialize_For_Incoming	CMINIC	Used by a program to initialize the conversation characteristics for an incoming conversation.
Prepare	CMPREP	Used by a program to prepare a subordinate for a commit operation.
Prepare_To_Receive	CMPTR	Used by a program to change a conversation from **Send** to **Receive** state in preparation to receive data.

Pseudonym	Call	Description
Receive	CMRCV	Used by a program to receive data.
Receive_Expedited_Data	CMRCVX	Used by a program to receive expedited data from its partner.
Release_Local_TP_Name	CMRLTP	Used by a program to release a name.
Request_To_Send	CMRTS	Used by a program to notify its partner that it would like to send data.
Send_Data	CMSEND	Used by a program to send data.
Send_Error	CMSERR	Used by a program to notify its partner of an error that occurred during the conversation.
Send_Expedited_Data	CMSNDX	Used by a program to send expedited data to its partner.
Set_AE_Qualifier	CMSAEQ	Used by a program to set the *AE_qualifier* conversation characteristic.
Set_Allocate_Confirm	CMSAC	Used by a program to set the *allocate_confirm* conversation characteristic.
Set_AP_Title	CMSAPT	Used by a program to set the *AP_title* conversation characteristic.
Set_Application_Context_Name	CMSACN	Used by a program to set the *application_context_name* conversation characteristic.
Set_Begin_Transaction	CMSBT	Used by a program to set the *begin_transaction* conversation characteristic.
Set_Confirmation_Urgency	CMSCU	Used by a program to set the *confirmation_urgency* conversation characteristic.
Set_Conversation_Security_Password	CMSCSP	Used by a program to set the *security_password* conversation characteristic.
Set_Conversation_Security_Type	CMSCST	Used by a program to set the *conversation_security_type* conversation characteristic.
Set_Conversation_Security_User_ID	CMSCSU	Used by a program to set the *security_user_ID* conversation characteristic.
Set _Conversation_Type	CMSCT	Used by a program to set the *conversation_type* conversation characteristic.

Pseudonym	Call	Description
Set_Deallocate_Type	CMSDT	Used by a program to set the *deallocate_type* conversation characteristic.
Set_Error_Direction	CMSED	Used by a program to set the *error_direction* conversation characteristic.
Set_Fill	CMSF	Used by a program to set the *fill* conversation characteristic.
Set_Initialization_Data	CMSID	Used by a program to set the *initialization_data* conversation characteristic.
Set_Join_Transaction	CMSJT	Used by a program to set the *join_transaction* conversation characteristic.
Set_Log_Data	CMSLD	Used by a program to set the *log_data* conversation characteristic.
Set_Mode_Name	CMSMN	Used by a program to set the *mode_name* conversation characteristic.
Set_Partner_LU_Name	CMSPLN	Used by a program to set the *partner_LU_name* conversation characteristic.
Set_Prepare_Data_Permitted	CMSPDP	Used by a program to set the *prepare_data_permitted* conversation characteristic.
Set_Prepare_To_Receive_Type	CMSPTR	Used by a program to set the *prepare_to_receive_type* conversation characteristic.
Set_Processing_Mode	CMSPM	Used by a program to set the *processing_mode* conversation characteristic.
Set_Queue_Callback_Function	CMSQCF	Used by a program to set a callback function, and a user field for a given conversation queue and to set the queue's processing mode to CM_NON_BLOCKING.
Set_Queue_Processing_Mode	CMSQPM	Used by a program to set the processing mode for a given conversation queue and to associate an outstanding-operation identifier (OOID) and a user field with the queue.
Set_Receive_Type	CMSRT	Used by a program to set the *receive_type* conversation characteristic.
Set_Return_Control	CMSRC	Used by a program to set the *return_control* conversation characteristic.

Pseudonym	Call	Description
Set_Send_Receive_Mode	CMSSRM	Used by a program to set the *send_receive_mode* conversation characteristic.
Set_Send_Type	CMSST	Used by a program to set the *send_type* conversation characteristic.
Set_Sync_Level	CMSSL	Used by a program to set the *sync_level* conversation characteristic.
Set_TP_Name	CMSTPN	Used by a program to set the *TP_name* conversation characteristic.
Set_Transaction_Control	CMSTC	Used by a program to set the *transaction_control* conversation characteristic.
Specify_Local_TP_Name	CMSLTP	Used by a program to associate a name with itself.
Test_Request_To_Send_Received	CMTRTS	Used by a program to determine whether or not the remote program is requesting to send data.
Wait_For_Completion	CMWCMP	Used by a program to wait for completion of one or more outstanding operations represented in a specified outstanding-operation-ID (OOID) list.
Wait_For_Conversation	CMWAIT	Used by a program to wait for the completion of any conversation-level outstanding operation.

NAME

Accept_Conversation (CMACCP) — accept an incoming conversation.

SYNOPSIS

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

DESCRIPTION

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 later allocates the conversation issues the Initialize_Conversation call, and the partner program that accepts the conversation after it is allocated issues the Accept_Conversation call.

The Accept_Conversation (CMACCP) call uses the following output 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_DEALLOCATED_ABEND
  This value indicates that CPI Communications deallocated the incoming conversation because an implicit call of *tx_set_transaction_control*( ) or *tx_begin*( ) failed.

  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

For half-duplex conversations, when *return_code* is set equal to CM_OK, the conversation enters **Receive** state.

For full-duplex conversations, when *return_code* is set equal to CM_OK, the conversation enters **Send-Receive** state.

APPLICATION USAGE

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 3-2 on page 30.

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. An implementation may choose to specify a minimum time before returning CM_PROGRAM_STATE_CHECK when no incoming conversation has arrived for the program.
6. 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.

SEE ALSO

Section 3.8 on page 29 provides a comparison of the conversation characteristics set by Initialize_For_Incoming, Initialize_Conversation and Accept_Conversation.

Section 4.2.1 on page 65 shows an example program flow using the Accept_Conversation call to accept a half-duplex conversation.

Section 4.3.9 on page 86 shows an example program flow using an Accept_Conversation call to accept a full-duplex conversation.

Initialize_Conversation (CMINIT) on page 195 describes how the conversation characteristics are initialized for the program that allocates the conversation.

NAME

Accept_Incoming (CMACCI) — accept an incoming conversation previously initialized with the Initialize_For_Incoming call.

SYNOPSIS

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

DESCRIPTION

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 one of the following calls:

CALL CMSJT – Set_Join_Transaction
CALL CMSPM – Set_Processing_Mode
CALL CMSQPM – Set_Queue_Processing_Mode
CALL CMSQCF – Set_Queue_Callback_Function.

The Accept_Incoming (CMACCI) call uses the following input and output 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_CALL_NOT_SUPPORTED

  CM_OPERATION_INCOMPLETE

  CM_DEALLOCATED_ABEND
  This value indicates that CPI Communications deallocated the incoming conversation because an implicit call of *tx_set_transaction_control*( ) or *tx_begin*( ) failed.

  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.

  — 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

For half-duplex conversations, when *return_code* is set to CM_OK, the conversation enters **Receive** state.

For full-duplex conversations, when *return_code* is set to CM_OK, the conversation enters **Send-Receive** state.

APPLICATION USAGE

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 3-2 on page 30 for a list of the conversation characteristics and how they are set by Initialize_For_Incoming and Accept_Incoming.
3. If Accept_Incoming is issued as a blocking call and no incoming conversation is available for the program, the call blocks until a conversation startup request arrives. The program can ensure that it is not placed in a wait state by taking one of the following actions before issuing the Accept_Incoming call:
   — For conversation-level non-blocking — set the *processing_mode* characteristic to CM_NON_BLOCKING by using the Set_Processing_Mode call
   — For queue-level non-blocking — set the processing mode for the Initialization queue to CM_NON_BLOCKING by using the Set_Queue_Callback_Function call or the Set_Queue_Processing_Mode call.
4. If the program has successfully issued a Set_Processing_Mode call, the subsequent Accept_Incoming call completes only when the conversation startup request is for a half-duplex conversation.
5. 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.
6. 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.

SEE ALSO

Section 3.8 on page 29 provides a comparison of the conversation characteristics set by Initialize_For_Incoming and Accept_Incoming.

Section 4.3.7 on page 82 and Section 4.3.8 on page 84 show example program flows using the Initialize_For_Incoming and Accept_Incoming calls.

Initialize_For_Incoming (CMINIC) on page 197 describes how the *conversation_ID* supplied on Accept_Incoming is assigned.

Set_Join_Transaction (CMSJT) on page 283 describes setting the *join_transaction* conversation characteristic.

Set_Processing_Mode (CMSPM) on page 295 describes setting the *processing_mode* conversation characteristic.

Set_Queue_Callback_Function (CMSQCF) on page 297 describes the how to set a callback function and related information for a non-blocking conversation queue.

Set_Queue_Processing_Mode (CMSQPM) on page 300 describes how to set the processing mode for a non-blocking conversation queue.

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

NAME

Allocate (CMALLC) — establish a conversation.

SYNOPSIS

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

DESCRIPTION

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

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

CALL CMSAEQ – Set_AE_Qualifier
CALL CMSAC – Set_Allocate_Confirm
CALL CMSAPT – Set_AP_Title
CALL CMSACN – Set_Application_Context_Name
CALL CMSBT – Set_Begin_Transaction
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 CMSID – Set_Initialization_Data
CALL CMSMN – Set_Mode_Name
CALL CMSPLN – Set_Partner_LU_Name
CALL CMSPM – Set_Processing_Mode
CALL CMSQCF – Set_Queue_Callback_Function
CALL CMSQPM – Set_Queue_Processing_Mode
CALL CMSRC – Set_Return_Control
CALL CMSSRM – Set_Send_Receive_Mode
CALL CMSSL – Set_Sync_Level
CALL CMSTPN – Set_TP_Name
CALL CMSTC – Set_Transaction_Control.

The Allocate (CMALLC) call uses the following input and output 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_RETRY_LIMIT_EXCEEDED
  This value indicates that the system-specified retry limit was exceeded.

  CM_SEND_RCV_MODE_NOT_SUPPORTED

  CM_SYNC_LVL_NOT_SUPPORTED_SYS

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 Section D.3.3 on page 481.)

— 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 *AP_title* characteristic (set from side information or using the Set_AP_Title call) or the *AE_qualifier* characteristic (set from side information or using the Set_AE_Qualifier call) or the *application_context_name* characteristic (set from side information or using the Set_Application_Context_Name call) specifies an AP title or an AE qualifier or an application context name that is not recognized as being valid.

— The *conversation_security_type* characteristic is CM_SECURITY_PROGRAM or CM_SECURITY_PROGRAM_STRONG, 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 or CM_SYNC_POINT_NO_CONFIRM and *transaction_control* set to CM_CHAINED_TRANSACTIONS or *begin_transaction* set to CM_BEGIN_IMPLICIT, the program is in the **Backout-Required** condition. New protected conversations cannot be allocated when the program is in this condition.

— *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, *transaction_control* is set to CM_CHAINED_TRANSACTIONS, and the program is not in transaction mode.

— The program has issued a successful Accept_Conversation (CMACCP) or Accept_Incoming (CMACCI) call on a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and using an OSI TP CRM, and the program has not issued a Receive (CMRCV) call on this conversation.

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

CM_OPERATION_NOT_ACCEPTED

CM_PRODUCT_SPECIFIC_ERROR

CM_SECURITY_NOT_SUPPORTED
: This value indicates that the requested conversation security type could not be provided. This is either because the remote system does not accept the requested type of security from the local system or because the requested security does not transport the type of required user name identified.

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 logical connection is not immediately available.

STATE CHANGES

For half-duplex conversations, when *return_code* is set to CM_OK, the conversation enters **Send** state.

For full-duplex conversations, when *return_code* is set to CM_OK, the conversation enters **Send-Receive** state.

APPLICATION USAGE

1. An allocation error resulting from the local system's failure to obtain a logical connection for the conversation is reported on the Allocate call. An allocation error resulting from the remote system's rejection of the conversation startup request is reported on a subsequent conversation call.
2. For CPI Communications to establish the conversation, CPI Communications must first establish a logical connection between the local system and the remote system, if such a connection does not already exist.
3. Depending on the circumstances, the local system can send the conversation startup request to the remote system as soon as it allocates a logical connection for the conversation. The local system can also buffer the conversation startup 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 logical connection allocated for the conversation and can vary from one logical connection 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 Section 3.11 on page 47 for more information on conversation security.
6. When *return_control* is set to CM_IMMEDIATE, the call completes immediately, regardless of the processing mode in effect for the Allocate call. If a logical connection is not available, *return_code* is set to CM_UNSUCCESSFUL.

7. Initialization data specified by use of the Set_Initialization_Data (CMSID) call is sent to the remote program along with the conversation startup request. The remote program may extract the initialization data with the Extract_Initialization_Data (CMEID) call.
8. By using the Set_Allocate_Confirm call, the program allocating the conversation may request notification that the remote program has confirmed its acceptance of the conversation.
9. If a conversation is using a particular CRM type, the Allocate call tries to establish a conversation using only the destination information for that CRM type.
10. If a program specifies destination information for both an OSI TP CRM and an LU 6.2 CRM but only one set of information is complete, the Allocate call tries only the destination for which CPI Communications has complete information. If complete destination information exists for use of both an OSI TP CRM and an LU 6.2 CRM, the Allocate call tries to establish a logical connection using one and then the other destination. Only if both attempts fail does the Allocate call return either CM_ALLOCATE_FAILURE_* or CM_UNSUCCESSFUL.

SEE ALSO

Section 4.2.1 on page 65 shows an example program flow using the Allocate call to establish a half-duplex conversation.

Section 4.3.1 on page 71 discusses control methods for data transmission.

Section 4.3.9 on page 86 shows an example program flow using an Allocate call to establish a full-duplex conversation.

Set_AE_Qualifier (CMSAEQ) on page 253 discusses the *AE_qualifier* conversation characteristic.

Set_Allocate_Confirm (CMSAC) on page 255 discusses the *allocate_confirm* conversation characteristic and explains an option for confirming acceptance of the conversation.

Set_AP_Title (CMSAPT) on page 257 discusses the *AP_title* conversation characteristic.

Set_Application_Context_Name (CMSACN) on page 259 discusses the *application_context_name* conversation characteristic.

Set_Begin_Transaction (CMSBT) on page 261 discusses the *begin_transaction* conversation characteristic.

Set_Conversation_Security_Password (CMSCSP) on page 265 discusses the *security_password* conversation characteristic.

Set_Conversation_Security_Type (CMSCST) on page 267 discusses the *conversation_security_type* conversation characteristic.

Set_Conversation_Security_User_ID (CMSCSU) on page 269 discusses the *security_user_ID* conversation characteristic.

Set_Conversation_Type (CMSCT) on page 271 discusses the *conversation_type* characteristic.

Set_Initialization_Data (CMSID) on page 281 and *Extract_Initialization_Data (CMEID)* on page 170 discuss the *initialization_data* conversation characteristic.

Set_Mode_Name (CMSMN) on page 287 discusses the *mode_name* conversation characteristic.

Set_Partner_LU_Name (CMSPLN) on page 289 discusses the *partner_LU_name* conversation characteristic.

Set_Processing_Mode (CMSPM) on page 295 describes setting the *processing_mode* conversation characteristic.

Set_Queue_Callback_Function (CMSQCF) on page 297 discusses how to set a callback function and related information for a conversation queue.

Set_Queue_Processing_Mode (CMSQPM) on page 300 discusses how to set the processing mode for a conversation queue.

Set_Return_Control (CMSRC) on page 305 discusses the *return_control* characteristic.

Set_Send_Receive_Mode (CMSSRM) on page 307 discusses how to set the send-receive mode for a conversation.

Set_Sync_Level (CMSSL) on page 311 discusses the *sync_level* conversation characteristic.

Set_TP_Name (CMSTPN) on page 313 discusses the *TP_name* conversation characteristic.

Set_Transaction_Control (CMSTC) on page 315 discusses the *transaction_control* conversation characteristic.

Section D.3.3 on page 481 discusses SNA service transaction programs.

NAME

Cancel_Conversation (CMCANC) — end a conversation immediately.

SYNOPSIS

```
CALL CMCANC(conversation_ID,return_code)
```

DESCRIPTION

A program uses Cancel_Conversation (CMCANC) to end a conversation immediately. Cancel_Conversation can be issued at any time, regardless of whether a previous operation is still in progress on the conversation.

Cancel_Conversation results in the immediate termination of any operations in progress on the specified conversation. No guarantees are given on the results of the terminated operations. For example, when a Cancel_Conversation call has been issued while a non-blocking Send_Data call is outstanding, the program cannot determine how much data was actually moved from the application buffer, nor can the program rely on the validity of any of the output parameters for the terminated Send_Data call.

The Cancel_Conversation (CMCANC) call uses the following input and output 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_CALL_NOT_SUPPORTED

  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.

APPLICATION USAGE

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 is 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 or CM_SYNC_POINT_NO_CONFIRM and backout is required, with the CM_DEALLOCATED_ABEND_BO return code.

   Note: For half-duplex conversations, if the conversation is using an LU 6.2 CRM and 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 240 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 program may be placed in the **Backout-Required** condition.
5. If the Cancel_Conversation call is the first operation on the conversation following an Accept (CMACCP) or Accept_Incoming (CMACCI) call and an OSI TP CRM is being used, then any initialization data specified by the use of the Set_Initialization_Data (CMSID) call is sent to the remote program.

SEE ALSO

Section 3.10 on page 43 discusses the use of non-blocking operations.

Set_Initialization_Data (CMSID) on page 281 and *Extract_Initialization_Data (CMEID)* on page 170 discuss the *initialization_data* conversation characteristic.

Wait_For_Conversation (CMWAIT) on page 325 and *Wait_For_Completion (CMWCMP)* on page 322 describe the normal completion of non-blocking operations.

NAME

Confirm (CMCFM) — send a confirmation request to its partner.

SYNOPSIS

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

DESCRIPTION

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.

Notes:

1. 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.
2. The Confirm call can be issued only on a half-duplex conversation.

The Confirm (CMCFM) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *control_information_received* (output)

  Specifies the variable containing an indication of whether or not control information has been received.

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

  CM_NO_CONTROL_INFO_RECEIVED
  : Indicates that no control information was received.

  CM_REQ_TO_SEND_RECEIVED (half-duplex conversations only)
  : 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 227. See the description of the Request_To_Send (CMRTS) call for further discussion of the local program's possible responses.

  CM_ALLOCATE_CONFIRMED (OSI TP CRM only)
  : The local program received confirmation of the remote program's acceptance of the conversation.

  CM_ALLOCATE_CONFIRMED_WITH_DATA (OSI TP CRM only)
  : The local program received confirmation of the remote program's acceptance of the conversation. The local program may now issue an Extract_Initialization_Data (CMEID) call to receive the initialization data.

  CM_ALLOCATE_REJECTED_WITH_DATA (OSI TP CRM only)
  : The remote program rejected the conversation. The local program may now issue an Extract_Initialization_Data (CMEID) call to receive the initialization data. This value is returned with a return code of CM_OK. The program receives a CM_DEALLOCATED_ABEND return code on a later call on the conversation.

  CM_EXPEDITED_DATA_AVAILABLE (LU 6.2 CRM only)
  : Expedited data is available to be received.

CM_RTS_RCVD_AND_EXP_DATA_AVAIL (half-duplex and LU 6.2 CRM only)
The local program received a request-to-send notification from the remote program and expedited data is available to be received.

Notes:

1. If *return_code* is set to CM_PROGRAM_PARAMETER_CHECK or CM_PROGRAM_STATE_CHECK, the value contained in *control_information_received* has no meaning.
2. When more than one piece of control information is available to be returned to the program, it is returned in the following order:
   - CM_ALLOCATE_CONFIRMED, CM_ALLOCATE_CONFIRMED_WITH_DATA or CM_ALLOCATE_REJECTED_WITH_DATA
   - CM_RTS_RCVD_AND_EXP_DATA_AVAIL
   - CM_REQ_TO_SEND_RECEIVED
   - CM_EXPEDITED_DATA_AVAILABLE
   - CM_NO_CONTROL_INFO_RECEIVED.

- *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_SYNC_LVL_NOT_SUPPORTED_SYS

  CM_SEND_RCV_MODE_NOT_SUPPORTED

  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**, **Send-Pending** or **Defer-Receive** 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 and *transaction_control* set to CM_CHAINED_TRANSACTIONS or *begin_transaction* set to CM_BEGIN_IMPLICIT, the program is in the **Backout-Required** condition. The Confirm call is not allowed for this conversation while the program is in this condition.

CM_PROGRAM_PARAMETER_CHECK
This value indicates one of the following:

— The *sync_level* conversation characteristic is set to CM_NONE or CM_SYNC_POINT_NO_CONFIRM.

— The *conversation_ID* specifies an unassigned conversation identifier.

— The *send_receive_mode* of the conversation is CM_FULL_DUPLEX.

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

CM_INCLUDE_PARTNER_REJECT_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.

APPLICATION USAGE

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 or the Cancel_Conversation call.

3. The send buffer of the local system is flushed as a result of this call.
4. When *control_information_received* indicates that expedited data is available, subsequent calls with this parameter continue to return the notification until the expedited data has been received.

SEE ALSO

Section 4.3.3 on page 74 shows an example program using the Confirm call.

Confirmed (CMCFMD) on page 141 provides information on the remote program's reply to the Confirm call.

Request_To_Send (CMRTS) on page 227 provides a complete discussion of the *control_information_received* parameter.

Set_Allocate_Confirm (CMSAC) on page 255 describes how a program can request that the remote program confirm its acceptance of the conversation.

Set_Sync_Level (CMSSL) on page 311 explains how programs specify the level of synchronization processing.

NAME

Confirmed (CMCFMD) — send a confirmation reply to its partner.

SYNOPSIS

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

DESCRIPTION

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.

A program can issue the Confirmed call on a full-duplex conversation only when deallocating a conversation that is using an OSI TP CRM.

The Confirmed (CMCFMD) call uses the following input and output 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:

  — For a half-duplex conversation, the conversation is not in **Confirm**, **Confirm-Send** or **Confirm-Deallocate** state.

  — For a full-duplex conversation, the conversation is not in **Confirm-Deallocate** state.

  — For a conversation with *sync_level* set to CM_SYNC_POINT, the program is in the **Backout-Required** condition. The Confirmed call is not allowed for this conversation while the program is in this condition.

  CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:

  — The *conversation_ID* specifies an unassigned conversation identifier.

  — The *send_receive_mode* is set to CM_FULL_DUPLEX and the conversation is using an LU 6.2 CRM.

  CM_OPERATION_NOT_ACCEPTED

  CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

For a half-duplex conversation, 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.

For a full-duplex conversation, when *return_code* is set to CM_OK, the conversation enters **Reset** state if the program received a *status_received* value of CM_CONFIRM_DEALLOC_RECEIVED on the preceding Receive call—that is, if the conversation was in **Confirm-Deallocate** state.

APPLICATION USAGE

1. For a half-duplex conversation, 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 waits until the local program responds with Confirmed.
2. For a half-duplex conversation, 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.

SEE ALSO

Section 4.3.3 on page 74 shows an example program using the Confirmed call.

Confirm (CMCFM) on page 137 provides more information on the Confirm call.

Receive (CMRCV) on page 208 provides more information on the *status_received* parameter.

Set_Sync_Level (CMSSL) on page 311 explains how programs specify the level of synchronization processing.

NAME

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

SYNOPSIS

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

DESCRIPTION

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.

The Convert_Incoming (CMCNVI) call uses the following input and output parameters:

- *buffer* (input) (output)

  Specifies the buffer containing the string to be converted. The contents of the string are 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_CALL_NOT_SUPPORTED

  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.

APPLICATION USAGE

1. When the EBCDIC hexadecimal codes, specified in Table A-2 on page 337, represent the encoding for the data transmitted across the network, the Convert_Incoming call can be used to convert 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 A-2 on page 337, 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.

SEE ALSO

Section 3.12 on page 48 provides information about data conversion and the Convert_Incoming call.

Convert_Outgoing (CMCNVO) on page 145 provides information about the Convert_Outgoing call.

NAME

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

SYNOPSIS

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

DESCRIPTION

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.

The Convert_Outgoing (CMCNVO) call uses the following input and output parameters:

- *buffer* (input) (output)

  Specifies the buffer containing the string to be converted. The contents of the string are 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_CALL_NOT_SUPPORTED

  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.

APPLICATION USAGE

1. When the EBCDIC hexadecimal codes, specified in Table A-2 on page 337, represent the encoding for the data transmitted across the network, the Convert_Outgoing call can be used to convert 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 A-2 on page 337, 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.

SEE ALSO

Section 3.12 on page 48 provides information about data conversion and the Convert_Outgoing call.

Convert_Incoming (CMCNVI) on page 143 provides information about the Convert_Incoming call.

NAME

Deallocate (CMDEAL) — end a conversation.

SYNOPSIS

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

DESCRIPTION

A program uses the Deallocate (CMDEAL) call to end a conversation. The *conversation_ID* is no longer assigned when the conversation is deallocated as part of this call.

For a half-duplex conversation, the deallocation can either be completed as part of this call or deferred until the program issues a resource recovery call. If the Deallocate call includes the function of the Flush or Confirm call, depending on the *deallocate_type* characteristic, the deallocation is completed as part of this call.

For a full-duplex conversation, the deallocation may be deferred until the program issues a resource recovery commit call. If the Deallocate call includes abnormal deallocation or the function of the Confirm call, depending on the *deallocate_type* characteristic, the deallocation is completed as part of this call. If the Deallocate call includes the function of the Flush call, depending on the *deallocate_type* characteristic, then the program can no longer send data to the partner. The deallocation is completed if the conversation was in **Send-Only** state before this call. Otherwise, the conversation goes to **Receive-Only** state. In this latter case, the deallocation is completed when a terminating error condition occurs, either this program or the partner program deallocates the conversation abnormally or cancels it, or the partner program deallocates the conversation using the function of the Flush 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.

The Deallocate (CMDEAL) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier of the conversation to be ended.

- *return_code* (output)

  Specifies the result of the call execution.

  Half-duplex Conversations

  The following return codes apply to half-duplex conversations:

  For any of the following conditions:

  — *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and either *sync_level* is set to CM_NONE or the conversation is in **Initialize-Incoming** state

  — *deallocate_type* is set to CM_DEALLOCATE_FLUSH

  — *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, but the conversation is not currently included in a transaction

  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 conversation is not in **Send**, **Send-Pending** or **Initialize-Incoming** state.

— The conversation is basic and in **Send** state; and the program started but did not finish sending a logical record.

— The *deallocate_type* is set to CM_DEALLOCATE_FLUSH, and the conversation is currently included in a transaction.

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_ABEND, the *return_code* variable can have one of the following values:

CM_OK (deallocation is completed)

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

CM_OPERATION_NOT_ACCEPTED

CM_OPERATION_INCOMPLETE

CM_PRODUCT_SPECIFIC_ERROR.

For any of the following conditions:

— *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and the *sync_level* is set to CM_CONFIRM

— *deallocate_type* is set to CM_DEALLOCATE_CONFIRM

— *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT, but the conversation is not currently included in a transaction

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_SYNC_LVL_NOT_SUPPORTED_SYS

CM_SEND_RCV_MODE_NOT_SUPPORTED

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 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 *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, and the conversation is currently included in a transaction.

— The *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, CM_DEALLOCATE_FLUSH or CM_DEALLOCATE_CONFIRM, the local program is the superior to the conversation, it has issued the Allocate (CMALLC) call with the *allocate_confirm* characteristic set to CM_ALLOCATE_CONFIRM, and it did not yet receive a *control_information_received* value of CM_ALLOCATE_CONFIRMED or CM_ALLOCATE_CONFIRMED_WITH_DATA.

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, *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and the conversation is included in a transaction, 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 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. Instead, the conversation is restored to the state it was in at the previous synchronization point. Table C-2 on page 398 and Table C-3 on page 400 show how 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 program is in the **Backout-Required** condition.

— The *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, CM_DEALLOCATE_FLUSH or CM_DEALLOCATE_CONFIRM, the local program is the superior to the conversation, it has issued the Allocate (CMALLC) call with the *allocate_confirm* characteristic set to CM_ALLOCATE_CONFIRM, and it did not yet receive a *control_information_received* value of CM_ALLOCATE_CONFIRMED or CM_ALLOCATE_CONFIRMED_WITH_DATA.

CM_PROGRAM_PARAMETER_CHECK
This value indicates one of the following:

— The *conversation_ID* specifies an unassigned conversation identifier.

— The conversation is using an OSI TP CRM, and the program is not the superior for the conversation.

CM_OPERATION_NOT_ACCEPTED

CM_PRODUCT_SPECIFIC_ERROR.

Full-duplex Conversations

The following return codes apply to full-duplex conversations:

For any of the following conditions:

— *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and either *sync_level* is set to CM_NONE or the conversation is in **Initialize-Incoming** state

— *deallocate_type* is set to CM_DEALLOCATE_FLUSH

— *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, but the conversation is not currently included in a transaction

the *return_code* variable can have one of the following values:

CM_OK
Deallocation is completed if this call was issued in **Send-Only** state. Otherwise, the call was issued in **Send-Receive** state and the conversation goes to **Receive-Only** state. In this latter case, the conversation will later get deallocated when a terminating error condition occurs, either this program or the partner program deallocates the conversation abnormally or cancels it, or the partner program deallocates the conversation using the function of the Flush call.

CM_PROGRAM_STATE_CHECK
This value indicates one of the following:

— The conversation is not in **Send-Receive** or **Send-Only** or **Initialize-Incoming** state.

— The conversation is basic and in **Send-Receive** or **Send-Only** state, and the program started but did not finish sending a logical record.

— The local program has received a *status_received* value of CM_JOIN_TRANSACTION and must issue a *tx_begin*( ) call to the TX (Transaction Demarcation) interface to join the transaction.

— The *deallocate_type* is set to CM_DEALLOCATE_FLUSH, and the conversation is currently included in a transaction.

— The *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, CM_DEALLOCATE_FLUSH or CM_DEALLOCATE_CONFIRM, the local program is the superior to the conversation, it has issued the Allocate (CMALLC) call with the *allocate_confirm* characteristic set to CM_ALLOCATE_CONFIRM, and it did not yet receive a *control_information_received* value of CM_ALLOCATE_CONFIRMED or CM_ALLOCATE_CONFIRMED_WITH_DATA.

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

CM_ALLOCATION_ERROR

CM_DEALLOCATED_ABEND

CM_DEALLOCATED_ABEND_SVC

CM_DEALLOCATED_ABEND_TIMER

CM_RESOURCE_FAILURE_NO_RETRY

CM_RESOURCE_FAILURE_RETRY

CM_DEALLOCATED_NORMAL

CM_OPERATION_NOT_ACCEPTED

CM_OPERATION_INCOMPLETE

CM_PRODUCT_SPECIFIC_ERROR.

If the *deallocate_type* conversation characteristic is set to CM_DEALLOCATE_ABEND, the *return_code* variable can have one of the following values:

CM_OK (deallocation is completed)

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

CM_OPERATION_NOT_ACCEPTED

CM_OPERATION_INCOMPLETE

CM_PRODUCT_SPECIFIC_ERROR.

If the *deallocate_type* conversation characteristic is set to CM_DEALLOCATE_CONFIRM, the *return_code* variable can have one of the following values:

CM_OK (deallocation is completed)

CM_DEALLOC_CONFIRM_REJECT
This value indicates that the partner program rejected the confirmation request. The conversation is not deallocated.

CM_ALLOCATION_ERROR

CM_DEALLOCATED_ABEND

CM_RESOURCE_FAILURE_NO_RETRY

CM_RESOURCE_FAILURE_RETRY

CM_DEALLOCATED_NORMAL

CM_PROGRAM_STATE_CHECK
This value indicates one of the following:

— The conversation is not in **Send-Receive** state.

— The conversation is basic and in **Send-Receive** state, and the program started but did not finish sending a logical record.

— The local program has received a *status_received* value of CM_JOIN_TRANSACTION and must issue a *tx_begin*( ) call to the TX (Transaction Demarcation) interface to join the transaction.

— The *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, and the conversation is currently included in a transaction.

— The *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, CM_DEALLOCATE_FLUSH or CM_DEALLOCATE_CONFIRM, the local program is the superior to the conversation, it has issued the Allocate (CMALLC) call with the *allocate_confirm* characteristic set to CM_ALLOCATE_CONFIRM, and it did not yet receive a *control_information_received* value of CM_ALLOCATE_CONFIRMED or CM_ALLOCATE_CONFIRMED_WITH_DATA.

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

CM_PRODUCT_SPECIFIC_ERROR

CM_OPERATION_INCOMPLETE

CM_OPERATION_NOT_ACCEPTED.

If the *deallocate_type* conversation characteristic is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, and the conversation is included in a transaction, 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 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. Instead, the conversation is restored to the state it was in at the previous synchronization point. Table C-2 on page 398 and Table C-3 on page 400 show how 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-Receive** state.

— The conversation is basic and in **Send-Receive** state, and the program started but did not finish sending a logical record.

— The program is in the **Backout-Required** condition.

— The local program has received a *status_received* value of CM_JOIN_TRANSACTION and must issue a *tx_begin*( ) call to the TX (Transaction Demarcation) interface to join the transaction.

— The *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, CM_DEALLOCATE_FLUSH or CM_DEALLOCATE_CONFIRM, the local program is the superior to the conversation, it has issued the Allocate (CMALLC) call with the *allocate_confirm* characteristic set to CM_ALLOCATE_CONFIRM, and it did not yet receive a *control_information_received* value of CM_ALLOCATE_CONFIRMED or CM_ALLOCATE_CONFIRMED_WITH_DATA.

CM_PROGRAM_PARAMETER_CHECK
This value indicates one of the following:

— The *conversation_ID* specifies an unassigned conversation identifier.

— The conversation is using an OSI TP CRM, and the program is not the superior for the conversation.

CM_DEALLOCATED_ABEND_BO

CM_DEALLOCATED_ABEND_SVC_BO

CM_DEALLOCATED_ABEND_TIMER_BO

CM_RESOURCE_FAILURE_RETRY_BO

CM_RESOURCE_FAIL_NO_RETRY_BO

CM_INCLUDE_PARTNER_REJECT_BO

CM_CONV_DEALLOC_AFTER_SYNCPT

CM_OPERATION_NOT_ACCEPTED

CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

For half-duplex conversations, when *return_code* indicates CM_OK:

- The conversation enters **Reset** state if *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and either *sync_level* is set to CM_NONE or the conversation is in **Initialize-Incoming** state, or if *deallocate_type* is set to one of the following:

  CM_DEALLOCATE_FLUSH

  CM_DEALLOCATE_CONFIRM

  CM_DEALLOCATE_ABEND.

- The conversation enters **Reset** state if *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, but the conversation is not currently included in a transaction.

- The conversation enters **Defer-Deallocate** state if *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and the conversation is included in a transaction.

For full-duplex conversations, when *return_code* indicates CM_OK:

- The conversation enters **Reset** state if *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and either *sync_level* is set to CM_NONE or the conversation is in **Initialize-Incoming** state, or if *deallocate_type* is set to CM_DEALLOCATE_CONFIRM or CM_DEALLOCATE_ABEND.
- The conversation enters **Reset** state if *deallocate_type* is set to CM_DEALLOCATE_FLUSH, or if *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM but the conversation is not currently included in a transaction, and the current state is **Send-Only** state.
- The conversation enters **Receive-Only** state if *deallocate_type* is set to CM_DEALLOCATE_FLUSH, or if *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM but the conversation is not currently included in a transaction, and the current state is **Send-Receive** state.
- The conversation enters **Defer-Deallocate** state if *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, and the conversation is included in a transaction.

APPLICATION USAGE

1. The execution of Deallocate includes the flushing of the local system's send buffer if any of the following conditions is true:

   — *deallocate_type* is set to CM_DEALLOCATE_FLUSH or CM_DEALLOCATE_CONFIRM

   — *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is CM_NONE or CM_CONFIRM

   — *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, but the conversation is not currently included in a transaction.

   If *deallocate_type* is CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and the conversation is included in a transaction, the local system's send buffer is not flushed until a resource recovery commit or backout call is issued by the program or the transaction manager.

2. If a conversation has *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, CPI Communications does not allow the conversation to be deallocated with a *deallocate_type* of CM_DEALLOCATE_CONFIRM or CM_DEALLOCATE_FLUSH unless *transaction_control* is set to CM_UNCHAINED_TRANSACTIONS and the conversation is not currently included in a transaction.

3. If *deallocate_type* is set to CM_DEALLOCATE_ABEND and 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.

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 local program issued Deallocate with the *deallocate_type* set to CM_DEALLOCATE_FLUSH, or with the *deallocate_type* set to

CM_DEALLOCATE_SYNC_LEVEL and *sync_level* set to CM_NONE, or with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* set to CM_SYNC_POINT_NO_CONFIRM, and the conversation not currently included in a transaction.

For a full-duplex conversation, this return code is returned on the Receive call, and if the conversation is using an OSI TP CRM, it is also returned on calls associated with the Send queue.

— CM_DEALLOCATED_ABEND *return_code*

This indicates that the local program issued Deallocate with *deallocate_type* set to CM_DEALLOCATE_ABEND.

For a full-duplex conversation, this return code is returned on the Receive call, and on some calls associated with the Send queue.

Note: For a half-duplex conversation, if the conversation is using an LU 6.2 CRM and 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 240 for a complete discussion.

— CM_DEALLOCATED_ABEND_BO *return_code*

This indicates that the local program issued Deallocate with the *deallocate_type* set to CM_DEALLOCATE_ABEND and with the conversation included in a transaction.

Note: For a half-duplex conversation, if the conversation is using an LU 6.2 CRM and 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 240 for a complete discussion.

— CM_CONFIRM_DEALLOC_RECEIVED *status_received* indication

This indicates that the local program issued Deallocate with the *deallocate_type* set to CM_DEALLOCATE_CONFIRM, or with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* set to CM_CONFIRM, or with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT, but with the conversation not currently included in a transaction.

— CM_TAKE_COMMIT_DEALLOCATE *status_received* indication

This indicates that the local 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 or CM_SYNC_POINT_NO_CONFIRM, and with the conversation included in a transaction.

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 the conversation is included in a transaction, the program may be placed in the **Backout-Required** condition.

7. If the conversation is using an OSI TP CRM and the Deallocate call with *deallocate_type* of CM_DEALLOCATE_ABEND is the first operation on the specified conversation following an Accept_Conversation (CMACCP) or Accept_Incoming (CMACCI) call, then any initialization data specified by use of the Set_Initialization_Data call is sent to the remote program.

8. If the Deallocate call on a full-duplex conversation is issued with the *deallocate_type* set to CM_DEALLOCATE_ABEND, the conversation is deallocated.

   If the conversation is not currently included in a transaction, outstanding calls associated with both the local and remote programs get return codes as follows:

   — Locally, all outstanding operations are terminated. No guarantees are given on the results of the terminated operations.

   — At the remote program:

      — New calls, other than Confirmed and Set_* calls, as well as outstanding calls, associated with the Send queue, get CM_DEALLOCATED_ABEND, and the conversation goes to **Receive-Only** or **Reset** state if it was in **Send-Receive** or **Send-Only** state, respectively.

      — Any data sent by the partner before it issued the Deallocate call can be received, after which the next Receive call will get CM_DEALLOCATED_ABEND. The conversation is now in **Reset** state.

      — Calls to the Expedited-Send queue until the conversation goes to **Reset** state get CM_CONVERSATION_ENDING. Outstanding calls are terminated when the conversation goes to **Reset** state. No guarantees are given on the results of the terminated operations.

      — Calls to the Expedited-Receive queue until the conversation goes to **Reset** state get CM_CONVERSATION_ENDING after any available expedited data has been received. Outstanding calls are terminated when the conversation goes to **Reset** state. No guarantees are given on the results of the terminated operations.

   If the conversation is currently included in a transaction, calls associated with the local Receive queue and the remote Receive queue, as well as certain calls associated with the remote Send queue, get CM_DEALLOCATED_ABEND_BO and the conversation goes to **Reset** state. Calls associated with the expedited data queues get the same return codes as when the conversation is not included in a transaction.

9. For a full-duplex conversation in **Send-Receive** state, when CM_DEALLOCATED_ABEND is returned to a call associated with the Send queue, the program can terminate the conversation by issuing Receive calls until it gets the CM_DEALLOCATED_ABEND return code that takes it to **Reset** state, or by issuing a Deallocate call with *deallocate_type* set to CM_DEALLOCATED_ABEND.

10. For a full-duplex conversation, if any of the following conditions is true:

    — *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is set to CM_NONE

    — *deallocate_type* is set to CM_DEALLOCATE_FLUSH

— *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, but the conversation is not currently included in a transaction

then the program can no longer send data on the conversation when a Deallocate call issued in **Send-Receive** state completes, and the *conversation_ID* is no longer assigned when a Deallocate call issued in **Send-Only** state completes.

If *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, or if *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is set to CM_SYNC_POINT but the conversation is not currently included in a transaction, then the *conversation_ID* is no longer assigned when the conversation is deallocated after confirmation.

If *deallocate_type* is CM_DEALLOCATE_ABEND, then the *conversation_ID* is no longer assigned when the conversation is deallocated as part of this call.

11. Implementors should note that a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND issued on 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.

SEE ALSO

Section 4.2.1 on page 65 shows an example program flow using the Deallocate call for a half-duplex conversation.

Section 4.3.11 on page 90 shows how a full-duplex conversation can be deallocated.

Set_Deallocate_Type (CMSDT) on page 273 discusses the *deallocate_type* characteristic and its possible values.

Set_Log_Data (CMSLD) on page 285 discusses the *log_data* characteristic.

NAME

Deferred_Deallocate (CMDFDE) — end a conversation following successful completion of the current transaction.

SYNOPSIS

```
CALL CMDFDE(conversation_ID,return_code)
```

DESCRIPTION

A program uses the Deferred_Deallocate (CMDFDE) call to end a conversation upon successful completion of the current transaction.

Deferred_Deallocate may be issued at any time during the transaction. Unlike the Deallocate call, it does not need to be the last call on the conversation. Deferred_Deallocate does not invalidate the conversation identifier.

Note: The Deferred_Deallocate (CMDFDE) call has meaning only when an OSI TP CRM is being used for the conversation.

The Deferred_Deallocate (CMDFDE) call uses the following input and output 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_CALL_NOT_SUPPORTED

  CM_OPERATION_INCOMPLETE

  CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:

  — This value indicates the *conversation_ID* specifies an unassigned identifier.

  — The conversation is not using an OSI TP CRM.

  — The program is not the superior for the conversation.

  CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:

  — The conversation is not in **Send** or **Send-Pending** state (for half-duplex conversations) or **Send-Receive** state (for full-duplex conversations).

  — The conversation is not currently included in a transaction.

  CM_TAKE_BACKOUT

  CM_DEALLOCATED_ABEND_BO

  CM_RESOURCE_FAILURE_RETRY_BO

  CM_RESOURCE_FAIL_NO_RETRY_BO

  CM_INCLUDE_PARTNER_REJECT_BO

  CM_OPERATION_NOT_ACCEPTED

CM_PRODUCT_SPECIFIC_ERROR.

Half-duplex Conversations

The following return codes apply to half-duplex conversations:

CM_SYNC_LVL_NOT_SUPPORTED_SYS

CM_SEND_RCV_MODE_NOT_SUPPORTED

CM_TPN_NOT_RECOGNIZED

CM_TP_NOT_AVAILABLE_NO_RETRY

CM_TP_NOT_AVAILABLE_RETRY

CM_PROGRAM_ERROR_PURGING.

Full-duplex Conversations

The following return code applies to full-duplex conversations:

CM_ALLOCATION_ERROR.

STATE CHANGES

This call does not cause an immediate state change. However, one of the following occurs after the transaction completes:

- If the transaction completes successfully, the conversation enters **Reset** state and the *conversation_ID* is no longer assigned.
- If the transaction does not complete successfully, the conversation returns to the state it was in at the beginning of the transaction, and the *conversation_ID* remains valid. If the conversation was initialized during the transaction, the conversation returns to the state it was in following conversation establishment.

APPLICATION USAGE

1. Once a commit call completes successfully for this transaction, a Deferred_Deallocate call performs the same function as a Deallocate (CMDEAL) call with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* set to either CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.
2. If the transaction does not complete successfully and is backed out, a previous Deferred_Deallocate call on the conversation is no longer in effect.

SEE ALSO

Deallocate (CMDEAL) on page 147 discusses conversation deallocation in more detail.

NAME

Extract_AE_Qualifier (CMEAEQ) — view the current *AE_qualifier* conversation characteristic.

SYNOPSIS

```
CALL CMEAEQ(conversation_ID,AE_qualifier,AE_qualifier_length,
            AE_qualifier_format,return_code)
```

DESCRIPTION

The local program uses the Extract_AE_Qualifier (CMEAEQ) call to extract the *AE_qualifier* conversation characteristic for a given conversation. The value is returned to the application in the *AE_qualifier* parameter.

The Extract_AE_Qualifier (CMEAEQ) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *AE_qualifier* (output)

  Specifies the variable containing the application-entity-qualifier for the remote program. The length of the variable must be at least 1024 bytes.

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

- *AE_qualifier_length* (output)

  Specifies the variable containing the length of the returned *AE_qualifier* parameter.

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

- *AE_qualifier_format* (output)

  Specifies the variable containing the format of the returned *AE_qualifier* parameter. The *AE_qualifier_format* variable can have one of the following values:

  CM_DN
  Specifies that the *AE_qualifier* is a distinguished name.

  CM_INT_DIGITS
  Specifies that the *AE_qualifier* is an integer represented as a sequence of decimal digits.

  Note: Unless *return_code* is set to CM_OK, the value of *AE_qualifier_format* 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_CALL_NOT_SUPPORTED

  CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned identifier.

  CM_PROGRAM_STATE_CHECK
  This value indicates that the *conversation_ID* specifies a conversation in **Initialize-Incoming** state.

CM_OPERATION_NOT_ACCEPTED
: This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.

CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause any state changes.

APPLICATION USAGE

1. This call does not change the *AE_qualifier*, *AE_qualifier_length* or *AE_qualifier_format* for the specified conversation.
2. The *AE_qualifier* may be either a distinguished name or an integer. Distinguished names may have any format and syntax that can be recognized by the local system. Integers are represented as a series of digits.
3. The call is not associated with any conversation queue. When a conversation uses queue-level non-blocking, the call does not return CM_OPERATION_NOT_ACCEPTED.

SEE ALSO

Set_AE_Qualifier (CMSAEQ) on page 253 and Section 3.5.2 on page 22 provide more information on the *AE_qualifier* conversation characteristic.

NAME

Extract_AP_Title (CMEAPT) — view the current *AP_title* conversation characteristic.

SYNOPSIS

```
CALL CMEAPT(conversation_ID,AP_title,AP_title_length,
                AP_title_format,return_code)
```

DESCRIPTION

A program uses the Extract_AP_Title (CMEAPT) call to extract the *AP_title* characteristic for a given conversation. The value is returned to the application in the *AP_title* parameter.

The Extract_AP_Title (CMEAPT) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *AP_title* (output)

  Specifies the variable containing the title of the application-process where the remote program is located. The length of the variable must be at least 1024 bytes.

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

- *AP_title_length* (output)

  Specifies the variable containing the length of the returned *AP_title* parameter.

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

- *AP_title_format* (output)

  Specifies the variable containing the format of the returned *AP_title* parameter. The *AP_title_format* variable can have one of the following values:

  CM_DN
  Specifies that the *AP_title* is a distinguished name.

  CM_OID
  Specifies that the *AP_title* is an object identifier.

  Note: Unless *return_code* is set to CM_OK, the value of *AP_title_format* 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_CALL_NOT_SUPPORTED

  CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned identifier.

  CM_PROGRAM_STATE_CHECK
  This value indicates that the *conversation_ID* specifies a conversation in **Initialize-Incoming** state.

  CM_OPERATION_NOT_ACCEPTED
  This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.

CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause any state changes.

APPLICATION USAGE

1. This call does not change the *AP_title*, *AP_title_length* or *AP_title_format* for the specified conversation.
2. The *AP_title* may be either a distinguished name or an object identifier. Distinguished names may have any format and syntax that can be recognized by the local system. Object identifiers are represented as a series of digits by periods (for example, *n.nn.n.nnn*).
3. The call is not associated with any conversation queue. When a conversation uses queue-level non-blocking, the call does not return CM_OPERATION_NOT_ACCEPTED on the conversation.

SEE ALSO

Set_AP_Title (CMSAPT) on page 257 and Section 3.5.2 on page 22 provide more information on the *AP_title* conversation characteristic.

NAME

Extract_Application_Context_Name (CMEACN) — view the current *application_context_name* conversation characteristic.

SYNOPSIS

```
CALL CMEACN(conversation_ID,application_context_name,
                application_context_name_length, return_code)
```

DESCRIPTION

Extract_Application_Context_Name (CMEACN) is used by a program to extract the *application_context_name* characteristic for a given conversation. The value is returned to the application in the *application_context_name* parameter.

The Extract_Application_Context_Name (CMEACN) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *application_context_name* (output)

  Specifies the application context name that is to be used on the conversation. The length of the variable must be at least 256 bytes.

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

- *application_context_name_length* (output)

  Specifies the length of the application context name that is to be used on the conversation.

  Note: Unless *return_code* is set to CM_OK, the value of *application_context_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_CALL_NOT_SUPPORTED

  CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned identifier.

  CM_PROGRAM_STATE_CHECK
  This value indicates that the *conversation_ID* specifies a conversation in **Initialize-Incoming** state.

  CM_OPERATION_NOT_ACCEPTED
  This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.

  CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause any state changes.

APPLICATION USAGE

1. This call does not change the *application_context_name* or the *application_context_name_length* conversation characteristic for the specified conversation.
2. The application context name is an object identifier and is represented as a series of digits separated by periods. For example, the default application context name defined by ISO for OSI TP with UDT is represented as 1.0.10026.6.2.
3. The call is not associated with any conversation queue. When a conversation uses queue-level non-blocking, the call does not return CM_OPERATION_NOT_ACCEPTED on the conversation.

SEE ALSO

Set_Application_Context_Name (CMSACN) on page 259 and Section 3.5.2 on page 22 provide more information on the *application_context_name* conversation characteristic.

NAME

Extract_Conversation_State (CMECS) — view the current state of a conversation.

SYNOPSIS

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

DESCRIPTION

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.

The Extract_Conversation_State (CMECS) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *conversation_state* (output)

  Specifies the conversation state that is returned to the local program.

  Half-duplex Conversations

  For half-duplex conversations, 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

  CM_PREPARED_STATE.

  Full-duplex Conversations

  For full-duplex conversations, the *conversation_state* can be one of the following:

  CM_INITIALIZE_STATE

  CM_CONFIRM_DEALLOCATE_STATE

  CM_DEFER_DEALLOCATE_STATE

  CM_SYNC_POINT_STATE

  CM_SYNC_POINT_DEALLOCATE_STATE

CM_INITIALIZE_INCOMING_STATE

CM_SEND_ONLY_STATE

CM_RECEIVE_ONLY_STATE

CM_SEND_RECEIVE_STATE

CM_PREPARED_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 or CM_SYNC_POINT_NO_CONFIRM, and the conversation is included in a transaction.

  — The conversation is not in **Initialize**, **Initialize-Incoming**, **Confirm-Deallocate**, **Send-Only** or **Receive-Only** state.

  — The program is in the **Backout-Required** condition.

  — The program is using protected resources that must be backed out.

  CM_OPERATION_NOT_ACCEPTED
  This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.

  CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause a state change.

APPLICATION USAGE

1. This call can be used to discover the state of a conversation after it has been backed out during a resource recovery backout operation.
2. The call is not associated with any conversation queue. When a conversation uses queue-level non-blocking, the call does not return CM_OPERATION_NOT_ACCEPTED on the conversation.

SEE ALSO

Section 3.14 on page 51 provides more information on using resource recovery interfaces.

NAME

Extract_Conversation_Type (CMECT) — view the current *conversation_type* conversation characteristic.

SYNOPSIS

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

DESCRIPTION

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.

The Extract_Conversation_Type (CMECT) call uses the following input and output 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
  : This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.

  CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause a state change.

APPLICATION USAGE

1. This call does not change the *conversation_type* for the specified conversation.
2. The call is not associated with any conversation queue. When a conversation uses queue-level non-blocking, the call does not return CM_OPERATION_NOT_ACCEPTED on the conversation.

SEE ALSO

Set_Conversation_Type (CMSCT) on page 271 provides more information on the *conversation_type* characteristic.

NAME

Extract_Initialization_Data (CMEID) — extract the current *initialization_data* conversation characteristic.

SYNOPSIS

```
CALL CMEID(conversation_ID,initialization_data,requested_length,
                initialization_data_length,return_code)
```

DESCRIPTION

A program uses the Extract_Initialization_Data (CMEID) call to extract the *initialization_data* and *initialization_data_length* conversation characteristics received from the remote program for a given conversation. The values are returned to the program in the *initialization_data* and *initialization_data_length* parameters.

The Extract_Initialization_Data call is used by the recipient of the conversation to extract the incoming initialization data received from the initiator of the conversation. It may be issued following the Accept_Conversation or Accept_Incoming call.

When the conversation is using an OSI TP CRM, the Extract_Initialization_Data call may also be used by the initiator of the conversation to extract the incoming initialization data from the recipient of the conversation. In this case, the call may be issued following receipt of a *control_information_received* value of CM_ALLOCATE_CONFIRMED_WITH_DATA or CM_ALLOCATE_REJECTED_WITH_DATA

The Extract_Initialization_Data (CMEID) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *initialization_data* (output)

  Specifies the variable containing the initialization data. Initialization data may be from 0 to 10000 bytes.

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

- *requested_length* (input)

  Specifies the length of the *initialization_data* variable to contain the initialization data.

- *initialization_data_length* (output)

  If *return_code* is CM_OK, the output value of this parameter specifies the size of the *initialization_data* variable in bytes. If *return_code* is CM_BUFFER_TOO_SMALL, this parameter indicates the size of the *initialization_data* to be extracted.

- *return_code* (output)

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

  CM_OK

  CM_CALL_NOT_SUPPORTED

CM_PROGRAM_PARAMETER_CHECK
This value indicates one of the following:

— The *conversation_ID* specifies an unassigned identifier.

— The *requested_length* specifies a value less than 0.

CM_BUFFER_TOO_SMALL
The *requested_length* specifies a value that is less than the size of the *initialization_data* characteristic to be extracted. The *initialization_data* characteristic is not returned, but the *initialization_data_length* parameter is set to indicate the size required.

CM_PROGRAM_STATE_CHECK
This value indicates that the *conversation_ID* specifies a conversation in **Initialize-Incoming** state.

CM_OPERATION_NOT_ACCEPTED
This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.

CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause any state changes.

APPLICATION USAGE

1. This call does not change the value of the *initialization_data* or the *initialization_data_length* characteristic for the specified conversation.
2. The program that initiates the conversation (issues Initialize_Conversation) must set *allocate_confirm* to CM_ALLOCATE_CONFIRM if it is expecting initialization data to be returned from the remote program following its confirmation of acceptance of the conversation.
3. The call is not associated with any conversation queue. When a conversation uses queue-level non-blocking, the call does not return CM_OPERATION_NOT_ACCEPTED on the conversation.

SEE ALSO

Set_Initialization_Data (CMSID) on page 281 describes how the initialization data is set by the remote program.

NAME

Extract_Maximum_Buffer_Size (CMEMBS) — extract the maximum buffer size supported by the system.

SYNOPSIS

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

DESCRIPTION

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

The Extract_Maximum_Buffer_Size (CMEMBS) call uses the following output 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.

APPLICATION USAGE

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 or extract in an Extract_Secondary_Information 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.

NAME

Extract_Mode_Name (CMEMN) — view the current *mode_name* conversation characteristic.

SYNOPSIS

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

DESCRIPTION

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.

The Extract_Mode_Name (CMEMN) call uses the following input and output 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 logical connection allocated, or to be allocated, which will carry the conversation specified by the *conversation_ID*. The length of the variable must be at least 8 bytes.

  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
  This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.

  CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause a state change.

APPLICATION USAGE

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.

3. The call is not associated with any conversation queue. When a conversation uses queue-level non-blocking, the call does not return CM_OPERATION_NOT_ACCEPTED on the conversation.

SEE ALSO

Section 3.8.5 on page 38 provides further information on the automatic conversion of the *mode_name* parameter.

Set_Mode_Name (CMSMN) on page 287 and Section 3.5.2 on page 22 provide further information on the *mode_name* characteristic.

NAME

Extract_Partner_LU_Name (CMEPLN) — view the current *partner_LU_name* conversation characteristic.

SYNOPSIS

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

DESCRIPTION

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.

The Extract_Partner_LU_Name (CMEPLN) call uses the following input and output 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. The length of the variable must be at least 17 bytes.

  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_CALL_NOT_SUPPORTED

  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
  This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.

  CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause a state change.

APPLICATION USAGE

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.
3. The call is not associated with any conversation queue. When a conversation uses queue-level non-blocking, the call does not return CM_OPERATION_NOT_ACCEPTED on the conversation.

SEE ALSO

Section 3.8.5 on page 38 provides further information on the automatic conversion of the *partner_LU_name* parameter.

Set_Partner_LU_Name (CMSPLN) on page 289 and Section 3.5.2 on page 22 provide more information on the *partner_LU_name* characteristic.

NAME

Extract_Secondary_Information (CMESI) — extract secondary information associated with the return code for a given call.

SYNOPSIS

```
CALL CMESI(conversation_ID,call_ID,buffer,requested_length,
           data_received,received_length,return_code)
```

DESCRIPTION

Extract_Secondary_Information (CMESI) is used to extract secondary information associated with the return code for a given call.

The Extract_Secondary_Information (CMESI) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *call_ID* (input)

  Specifies the call identifier (see Table A-1 on page 330).

- *buffer* (output)

  Specifies the variable in which the program is to receive the secondary information.

  Note: *buffer* contains information only if the *return_code* is set to CM_OK.

- *requested_length* (input)

  Specifies the maximum amount of secondary information the program is to receive. Valid *requested_length* values range from 1 to the maximum buffer size supported by the system. The maximum buffer size is at least 32767 bytes. See the **APPLICATION USAGE** section of *Receive (CMRCV)* on page 208 for additional information about determining the maximum buffer size.

- *data_received* (output)

  Specifies whether or not the program received complete secondary information.

  Note: Unless *return_code* is set to CM_OK, the value contained in *data_received* has no meaning.

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

  CM_COMPLETE_DATA_RECEIVED
  : This value indicates that complete secondary information is received.

  CM_INCOMPLETE_DATA_RECEIVED
  : This value indicates that more secondary information is available to be received.

- *received_length* (output)

  Specifies the variable containing the amount of secondary information the program received, up to the maximum.

  Note: Unless *return_code* is set to CM_OK, the value contained in *received_length* has no meaning.

- *return_code* (output)

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

  CM_OK

  CM_CALL_NOT_SUPPORTED

  CM_NO_SECONDARY_INFORMATION
  This value indicates that no secondary information is available for the specified call on the specified conversation.

  CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:

  — The *conversation_ID* specifies an unassigned conversation identifier.

  — The *call_ID* specifies CM_CMESI or an undefined value.

  — The *requested_length* specifies a value that exceeds the range permitted by the implementation.

STATE CHANGES

This call does not cause a state change.

APPLICATION USAGE

1. The program should issue the call immediately after it receives a return code at the completion of a call. In particular, when a conversation is deallocated and enters **Reset** state, the associated secondary information and conversation identifier are available for a system-dependent time.
2. If an Accept_Conversation, Initialize_Conversation, or Initialize_For_Incoming call fails, a conversation identifier is assigned and returned to the program for use only on the Extract_Secondary_Information call.
3. When the Extract_Secondary_Information call completes successfully, CPI Communications no longer keeps the returned secondary information for the specified call on the specified conversation. The same information is not available for a subsequent Extract_Secondary_Information call.
4. The program cannot use the call to retrieve secondary information for the previous Extract_Secondary_Information call.
5. When the *call_ID* specifies one of the non-conversation calls (i.e., Convert_Incoming, Convert_Outgoing, Extract_Maximum_Buffer_Size, Release_Local_TP_Name, Specify_Local_TP_Name, Wait_For_Conversation and Wait_For_Completion), the *conversation_ID* is ignored.
6. Implementors should note that because of different non-blocking levels, an implementation should maintain secondary information as follows:

   — For conversations using conversation-level non-blocking, secondary information is kept:

   — on a per-conversation basis for conversation calls

   — on a per-thread basis for non-conversation calls.

— For conversations not using conversation-level non-blocking, secondary information is kept:

  — on a per-queue basis for calls that are associated with a conversation queue

  — on a per-thread basis for calls that are not associated with any conversation queue.

SEE ALSO

Extract_Maximum_Buffer_Size (CMEMBS) on page 172 further discusses determining the maximum buffer size supported by the system.

Section B.2 on page 362 provides a complete discussion of secondary information.

NAME

Extract_Security_User_ID (CMESUI) — view the current *security_user_ID* conversation characteristic.

SYNOPSIS

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

DESCRIPTION

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

The Extract_Security_User_ID (CMESUI) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *security_user_ID* (output)

  Specifies the variable containing the user ID. The length of the variable must be at least 10 bytes.

  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_CALL_NOT_SUPPORTED

  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
  : This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.

  CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause any state changes.

APPLICATION USAGE

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.

3. The call is not associated with any conversation queue. When a conversation uses queue-level non-blocking, the call does not return CM_OPERATION_NOT_ACCEPTED on the conversation.

SEE ALSO

Section 3.8.5 on page 38 provides further information on the automatic conversion of the *security_user_ID* parameter.

Set_Conversation_Security_User_ID (CMSCSU) on page 269 discusses the setting of the *security_user_ID* characteristic.

NAME

Extract_Send_Receive_Mode (CMESRM) — view the current *send_receive_mode* conversation characteristic.

SYNOPSIS

```
CALL CMESRM(conversation_ID,send_receive_mode,return_code)
```

DESCRIPTION

The Extract_Send_Receive_Mode (CMESRM) call is used by a program to extract the value of the *send_receive_mode* characteristic for a conversation. The value is returned in the *send_receive_mode* parameter.

The Extract_Send_Receive_Mode (CMESRM) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *send_receive_mode* (output)

  Specifies the send-receive mode for the conversation.

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

  CM_HALF_DUPLEX
  : Indicates that the conversation is a half-duplex conversation.

  CM_FULL_DUPLEX
  : Indicates that the conversation is a full-duplex conversation.

  Note: Unless *return_code* is set to CM_OK, the value of *send_receive_mode* 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_CALL_NOT_SUPPORTED

  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
  : This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.

  CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause a state change.

APPLICATION USAGE

1. This call does not change the *send_receive_mode* for the specified conversation.
2. This call is not associated with any conversation queue. When a conversation uses queue-level non-blocking, the call does not return CM_OPERATION_NOT_ACCEPTED on the conversation.

SEE ALSO

Section 3.3 on page 19 provides more information on the differences between half-duplex and full-duplex conversations.

Section 4.3.9 on page 86 shows an example of how a full-duplex conversation is set up.

Set_Send_Receive_Mode (CMSSRM) on page 307 describes how to set the *send_receive_mode* characteristic for a conversation.

NAME

Extract_Sync_Level (CMESL) — view the current *sync_level* conversation characteristic.

SYNOPSIS

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

DESCRIPTION

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.

The Extract_Sync_Level (CMESL) call uses the following input and output 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 (half-duplex conversations only)
  : 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 (half-duplex conversations only)
  : 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.

  CM_SYNC_POINT_NO_CONFIRM
  : For systems that support resource recovery processing, this value specifies that the conversation is a protected resource. The programs can issue resource recovery interface calls and will recognize returned parameters relating to resource recovery processing. The programs cannot perform confirmation processing.

  Notes:

  1. Unless *return_code* is set to CM_OK, the value of *sync_level* is not meaningful.
  2. If the conversation is using an OSI TP CRM, confirmation of the deallocation of the conversation can be performed with any *sync_level* value.

- *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
: This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.

CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause a state change.

APPLICATION USAGE

1. This call does not change the *sync_level* for the specified conversation.
2. The call is not associated with any conversation queue. When a conversation uses queue-level non-blocking, the call does not return CM_OPERATION_NOT_ACCEPTED on the conversation.

SEE ALSO

Set_Sync_Level (CMSSL) on page 311 provides more information on the *sync_level* characteristic.

NAME

Extract_TP_Name (CMETPN) — determine the *TP_name* characteristic's value for a given conversation.

SYNOPSIS

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

DESCRIPTION

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.

The Extract_TP_Name (CMETPN) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *TP_name* (output)

  Specifies the variable containing the *TP_name* for the specified conversation. The length of the variable must be at least 64 bytes.

  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_CALL_NOT_SUPPORTED

  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
  : This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.

  CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause any state changes.

APPLICATION USAGE

1. 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.
2. The call is not associated with any conversation queue. When a conversation uses queue-level non-blocking, the call does not return CM_OPERATION_NOT_ACCEPTED on the conversation.

SEE ALSO

Specify_Local_TP_Name (CMSLTP) on page 317 provides more information on handling multiple names within a single program.

NAME

Extract_Transaction_Control (CMETC) — extract the *transaction_control* characteristic's value for a given conversation.

SYNOPSIS

```
CALL CMETC(conversation_ID,transaction_control,return_code)
```

DESCRIPTION

Extract_Transaction_Control (CMETC) is used by a program to extract the *transaction_control* characteristic for a given conversation. The value is returned to the application program in the *transaction_control* parameter.

The *transaction_control* characteristic is used only by an OSI TP CRM and is not significant unless the *sync_level* characteristic is set to either CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.

The Extract_Transaction_Control (CMETC) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *transaction_control* (output)

  Specifies the variable containing the *transaction_control* characteristic for the specified conversation. The *transaction_control* variable can have one of the following values:

  CM_CHAINED_TRANSACTIONS
  : Specifies that the conversation uses chained transactions.

  CM_UNCHAINED_TRANSACTIONS
  : Specifies that the conversation uses unchained transactions.

  Note: Unless *return_code* is set to CM_OK, the value of *transaction_control* 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_CALL_NOT_SUPPORTED

  CM_PROGRAM_PARAMETER_CHECK
  : This value indicates that the *conversation_ID* specifies an unassigned identifier.

  CM_PROGRAM_STATE_CHECK
  : This value indicates that the *conversation_ID* specifies a conversation in **Initialize-Incoming** state.

  CM_OPERATION_NOT_ACCEPTED
  : This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.

  CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause any state changes.

APPLICATION USAGE

1. This call does not change the *transaction_control* for the specified conversation.
2. This call is not associated with any conversation queue. When a conversation uses queue-level non-blocking, the call does not return CM_OPERATION_NOT_ACCEPTED.
3. This call can be used by the recipient to determine the *transaction_control* characteristic for the conversation.

   If the value is CM_CHAINED_TRANSACTIONS and *join_transaction* is set to CM_JOIN_IMPLICIT, the conversation is already included in a transaction. If the value is CM_CHAINED_TRANSACTIONS and *join_transaction* is set to CM_JOIN_EXPLICIT, the program must issue a *tx_begin*( ) call to join the transaction.

   If the value is CM_UNCHAINED_TRANSACTIONS, the program is informed with a CM_JOIN_TRANSACTION *status_received* value if it is to join the transaction. If the *join_transaction* characteristic is set to CM_JOIN_IMPLICIT, the conversation is already included in the transaction. If the *join_transaction* characteristic is set to CM_JOIN_EXPLICIT, the program must issue a *tx_begin*( ) call to join the transaction.

SEE ALSO

Set_Transaction_Control (CMSTC) on page 315 provides more information on the *transaction_control* characteristic.

NAME

Flush (CMFLUS) — flush the local CRM's send buffer.

SYNOPSIS

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

DESCRIPTION

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.

The Flush (CMFLUS) call uses the following input and output 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_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation ID.

  CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:

  — For a half-duplex conversation, the conversation is not in **Send**, **Send-Pending**, or **Defer-Receive** state.

  — For a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the program is in the **Backout-Required** condition. The Flush call is not allowed for this conversation while the program is in this condition.

  CM_OPERATION_NOT_ACCEPTED

  CM_PRODUCT_SPECIFIC_ERROR.

  Full-duplex Conversations

  The following return codes apply to full-duplex conversations.

  If the conversation is not currently included in a transaction, the *return_code* can have one of the following values:

  CM_ALLOCATION_ERROR

  CM_DEALLOCATED_ABEND

  CM_DEALLOCATED_ABEND_SVC

  CM_DEALLOCATED_ABEND_TIMER

  CM_RESOURCE_FAILURE_NO_RETRY

  CM_RESOURCE_FAILURE_RETRY

CM_DEALLOCATED_NORMAL

CM_PROGRAM_STATE_CHECK

This value indicates that the program has received a *status_received* value of CM_JOIN_TRANSACTION and must issue a *tx_begin*( ) call to the TX (Transaction Demarcation) interface to join the transaction.

If the *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM and the conversation is included in a transaction, the *return_code* can have one of the following values:

CM_TAKE_BACKOUT

CM_DEALLOCATED_ABEND_BO

CM_DEALLOCATED_ABEND_SVC_BO

CM_DEALLOCATED_ABEND_TIMER_BO

CM_RESOURCE_FAILURE_RETRY_BO

CM_RESOURCE_FAIL_NO_RETRY_BO

CM_INCLUDE_PARTNER_REJECT_BO

CM_CONV_DEALLOC_AFTER_SYNCPT.

STATE CHANGES

For half-duplex conversations, 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.

For full-duplex conversations, this call does not cause any state changes.

APPLICATION USAGE

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. The use of Send_Data followed by a call to Flush is equivalent to the use of Send_Data after setting *send_type* to CM_SEND_AND_FLUSH.

4. For full-duplex conversations, when CM_ALLOCATION_ERROR, CM_DEALLOCATE_*, CM_RESOURCE_FAILURE_* or CM_DEALLOCATE_NORMAL is returned and the conversation is in **Send-Receive** state, the program can terminate the conversation by issuing Receive calls until it gets a return code that takes it to **Reset** state, or by issuing a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND.

SEE ALSO

Section 4.3.1 on page 71 discusses the conditions for data transmission.

Section 4.3.3 on page 74 shows an example of how a program can use the Flush call to establish a conversation immediately.

Allocate (CMALLC) on page 130 provides more information on how information is buffered from the Allocate call.

Send_Data (CMSEND) on page 230 provides more information on how information is buffered from the Send_Data call.

Send_Error (CMSERR) on page 240 provides more information on how information is buffered from the Send_Error call.

Set_Send_Type (CMSST) on page 309 discusses alternative methods of achieving the Flush function.

NAME

Include_Partner_In_Transaction (CMINCL) — include a partner program in a transaction.

SYNOPSIS

```
CALL CMINCL(conversation_ID,return_code)
```

DESCRIPTION

A program uses the Include_Partner_In_Transaction (CMINCL) call to explicitly request that the subordinate join the transaction. The caller must be in a transaction, and the subordinate must be on a branch supporting unchained transactions.

Note: The Include_Partner_In_Transaction call has meaning only when an OSI TP CRM is being used for the conversation.

The Include_Partner_In_Transaction (CMINCL) call uses the following input and output 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_CALL_NOT_SUPPORTED

  CM_OPERATION_INCOMPLETE

  CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:

  — The *conversation_ID* specifies an unassigned conversation identifier.

  — The conversation is not using an OSI TP CRM.

  — The *transaction_control* is CM_CHAINED_TRANSACTIONS.

  — The program is not the superior for the conversation.

  CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:

  — For a half-duplex conversation, the conversation is not in **Send** or **Send-Pending** state.

  — For a full-duplex conversation, the conversation is not in **Send-Receive** state.

  — The conversation is basic and in **Send** state (for a half-duplex conversation) or **Send-Receive** state (for a full-duplex conversation), and the program started but did not finish sending a logical record.

  — The program is not in transaction mode. The program must issue a *tx_begin*( ) call to the TX (Transaction Demarcation) interface to start a transaction.

  — The conversation is already included in the current transaction.

  CM_DEALLOCATED_ABEND

  CM_RESOURCE_FAILURE_NO_RETRY

CM_RESOURCE_FAILURE_RETRY

CM_OPERATION_NOT_ACCEPTED

CM_PRODUCT_SPECIFIC_ERROR.

Half-duplex Conversations

The following return codes apply to half-duplex conversations:

CM_SEND_RCV_MODE_NOT_SUPPORTED

CM_SYNC_LVL_NOT_SUPPORTED_SYS

CM_TPN_NOT_RECOGNIZED

CM_TP_NOT_AVAILABLE_NO_RETRY

CM_TP_NOT_AVAILABLE_RETRY

CM_PROGRAM_ERROR_PURGING.

Full-duplex Conversations

The following return codes apply to full-duplex conversations:

CM_ALLOCATION_ERROR

CM_DEALLOCATED_NORMAL.

STATE CHANGES

This call does not cause any state changes.

APPLICATION USAGE

1. The call is used by a program to request that the subordinate join the transaction when the *begin_transaction* conversation characteristic is set to CM_BEGIN_EXPLICIT.
2. The remote program receives the request to join the transaction as a *status_received* indicator of CM_JOIN_TRANSACTION on a Receive call.

SEE ALSO

Section 3.14.5 on page 58 discusses chained and unchained transactions.

Section 3.14.6 on page 59 discusses how a program requests the partner program to join a transaction.

Set_Begin_Transaction (CMSBT) on page 261 discusses the *begin_transaction* characteristic.

Set_Join_Transaction (CMSJT) on page 283 discusses how the remote program receives the request to join the transaction.

NAME

Initialize_Conversation (CMINIT) — initialize the conversation characteristics for an outgoing conversation.

SYNOPSIS

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

DESCRIPTION

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 or the Initialize_Incoming and Accept_Incoming calls 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.

The Initialize_Conversation (CMINIT) call uses the following input and output 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. 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_PROGRAM_PARAMETER_CHECK
  This value indicates that the *sym_dest_name* specifies an unrecognized value.

  CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

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

APPLICATION USAGE

1. For a list of the conversation characteristics that are initialized when the Initialize_Conversation call completes successfully, see Table 3-2 on page 30.

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 Section 3.7.1 on page 26.

4. If the side information 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. 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.

SEE ALSO

Section 3.5.2 on page 22 provides more information on *sym_dest_name*.

Section 3.8 on page 29 provides a general overview of conversation characteristics and how they are used by the program and CPI Communications.

Section 4.2.1 on page 65 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.

NAME

Initialize_For_Incoming (CMINIC) — initialize the conversation characteristics for an incoming conversation.

SYNOPSIS

```
CALL CMINIC(conversation_ID,return_code)
```

DESCRIPTION

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.

The Initialize_For_Incoming (CMINIC) call uses the following output 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_CALL_NOT_SUPPORTED

  CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

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

APPLICATION USAGE

1. For a list of the conversation characteristics initialized when the Initialize_For_Incoming call completes successfully, see Table 3-2 on page 30.
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 3-2 on page 30, 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.

SEE ALSO

Section 3.8 on page 29 describes how the conversation characteristics are initialized by the Initialize_For_Incoming and Accept_Incoming calls.

Section 4.3.7 on page 82 and Section 4.3.8 on page 84 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.

Accept_Incoming (CMACCI) on page 127 describes how the *conversation_ID* is used when an incoming conversation is accepted by a program.

NAME

Prepare (CMPREP) — prepare a subordinate for a commit operation.

SYNOPSIS

```
CALL CMPREP(conversation_ID,return_code)
```

DESCRIPTION

A program uses the Prepare (CMPREP) call to explicitly request that a branch of the transaction prepare its resources to commit changes made during the transaction.

When the conversation is using an OSI TP CRM and the caller is not the root of the transaction, the caller must have received a take-commit notification from its superior. The subordinate program on the conversation cannot issue a Prepare (CMPREP) call.

The Prepare (CMPREP) call uses the following input and output 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_CALL_NOT_SUPPORTED

  CM_OPERATION_INCOMPLETE

  CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:

  — The *conversation_ID* specifies an unassigned conversation identifier.

  — The *sync_level* is not CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.

  — The conversation is using an OSI TP CRM, and the program is not the superior for the conversation.

  CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:

  — For a half-duplex conversation, the conversation is not in **Send**, **Send-Pending**, **Defer-Receive** or **Defer-Deallocate** state.

  — For a full-duplex conversation, the conversation is not in **Send-Receive** state.

  — The conversation is basic, and the program started but did not finish sending a logical record.

  — The program is not in transaction mode. The program must issue a *tx_begin*( ) call to the TX (Transaction Demarcation) interface to start a transaction.

  — For a conversation with *transaction_control* set to CM_CHAINED_TRANSACTIONS or *begin_transaction* set to CM_BEGIN_IMPLICIT, the program is in the **Backout-Required** condition.

  — The conversation is using an OSI TP CRM, *begin_transaction* is set to CM_BEGIN_EXPLICIT, and the conversation is not currently included in a transaction.

— The conversation is using an OSI TP CRM, and the program is not the root of the transaction and has not received a take-commit notification from its superior.

CM_RESOURCE_FAILURE_RETRY

CM_OPERATION_NOT_ACCEPTED

CM_TAKE_BACKOUT

CM_DEALLOCATED_ABEND_BO

CM_DEALLOCATED_ABEND_SVC_BO

CM_DEALLOCATED_ABEND_TIMER_BO

CM_RESOURCE_FAILURE_RETRY_BO

CM_RESOURCE_FAIL_NO_RETRY_BO

CM_INCLUDE_PARTNER_REJECT_BO

CM_PRODUCT_SPECIFIC_ERROR.

Half-duplex Conversations

The following return codes apply to half-duplex conversations:

CM_CONVERSATION_TYPE_MISMATCH

CM_PIP_NOT_SPECIFIED_CORRECTLY

CM_SECURITY_NOT_VALID

CM_SYNC_LVL_NOT_SUPPORTED_PGM

CM_SYNC_LVL_NOT_SUPPORTED_SYS

CM_SEND_RCV_MODE_NOT_SUPPORTED

CM_TPN_NOT_RECOGNIZED

CM_TP_NOT_AVAILABLE_NO_RETRY

CM_TP_NOT_AVAILABLE_RETRY

CM_PROGRAM_ERROR_PURGING.

Full-duplex Conversations

The following return code applies to full-duplex conversations:

CM_ALLOCATION_ERROR

CM_CONV_DEALLOC_AFTER_SYNCPT.

STATE CHANGES

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

APPLICATION USAGE

1. The Prepare (CMPREP) functions without waiting for a response from the remote program. A program can detect that its partner has prepared its transaction resources by issuing a Receive (CMRCV) call and checking the result in *status_received*, or the caller can complete the transaction without waiting for the partner to respond.

2. A program cannot send data to the remote program after issuing a Prepare call.
3. A program that issues the Prepare call may receive data from the remote program if the conversation is using an OSI TP CRM and either *prepare_data_permitted* is set to CM_PREPARE_DATA_PERMITTED or the conversation is full-duplex.
4. The partner finds out about the Prepare call by receiving one of the take-commit notifications described in Table 3-7 on page 53 (half-duplex) or Table 3-8 on page 54 (full-duplex).

SEE ALSO

Receive (CMRCV) on page 208 discusses the *status_received* parameter.

Set_Prepare_Data_Permitted (CMSPDP) on page 291 discusses the *prepare_data_permitted* characteristic.

NAME

Prepare_To_Receive (CMPTR) — change a conversation from **Send** to **Receive** state in preparation to receive data.

SYNOPSIS

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

DESCRIPTION

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.

Note: The Prepare_To_Receive_Type call can be issued only on a half-duplex conversation.

The Prepare_To_Receive (CMPTR) call uses the following input and output 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.

  For any of the following conditions:

  — *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

  — *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, but the conversation is not currently included in a transaction

  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 **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 or CM_SYNC_POINT_NO_CONFIRM and *transaction_control* set to CM_CHAINED_TRANSACTIONS or *begin_transaction* set to CM_BEGIN_IMPLICIT, the program is in the **Backout-Required** condition. The Prepare_To_Receive call is not allowed for this conversation while the program is in this condition.

CM_PROGRAM_PARAMETER_CHECK
This value indicates one of the following:

— The *conversation_ID* specifies an unassigned conversation identifier.

— The *send_receive_mode* of the conversation is CM_FULL_DUPLEX.

CM_OPERATION_NOT_ACCEPTED

CM_PRODUCT_SPECIFIC_ERROR.

For any of the following conditions:

— *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_SYNC_LEVEL and *sync_level* is set to CM_SYNC_POINT, but the conversation is not currently included in a transaction

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_SYNC_LVL_NOT_SUPPORTED_SYS

CM_SEND_RCV_MODE_NOT_SUPPORTED

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 or CM_SYNC_POINT_NO_CONFIRM and *transaction_control* set to CM_CHAINED_TRANSACTIONS or *begin_transaction* set to CM_BEGIN_IMPLICIT, the program 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

CM_INCLUDE_PARTNER_REJECT_BO.

If *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, *sync_level* is set to is CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and the conversation is included in a transaction, 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 **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 program is in the **Backout-Required** condition.

— A prior call to Deferred_Deallocate is still in effect for the conversation.

CM_PROGRAM_PARAMETER_CHECK
This value indicates one of the following:

— The *conversation_ID* specifies an unassigned conversation identifier.

— The *send_receive_mode* of the conversation is CM_FULL_DUPLEX.

— The conversation is using an OSI TP CRM, and the program is not the superior for the conversation.

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_FLUSH.
  - 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_NONE or CM_CONFIRM.
  - The *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, but the conversation is not currently included in a transaction.
- The conversation enters the **Defer-Receive** state if *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and the conversation is included in a transaction.

APPLICATION USAGE

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, or if *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* is CM_SYNC_POINT but the conversation is not currently included in a transaction, 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, or if *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM but the conversation is not currently included in a transaction, the system's send buffer is flushed if it contains information, and send control is transferred to the remote program without any synchronizing acknowledgement.
   - If *sync_level* is set to CM_CONFIRM, or if *sync_level* is set to CM_SYNC_POINT but the conversation is not currently included in a transaction, 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 or CM_SYNC_POINT_NO_CONFIRM and the conversation is included in a transaction, 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 acknowledgement. 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 one of the following:

   — *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_FLUSH

   — *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_NONE

   — *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT_NO_CONFIRM, but with the conversation not currently included in a transaction.

   CM_CONFIRM_SEND_RECEIVED
   The local program issued this call with one of the following:

   — *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_CONFIRM

   — *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_CONFIRM

   — *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT, but with the conversation not currently included in a transaction.

   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 or CM_SYNC_POINT_NO_CONFIRM, and with the conversation included in a transaction.

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. When the conversation is using an OSI TP CRM and the Deferred_Deallocate call has been issued on the conversation, the Prepare_To_Receive call with *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL is not allowed. The call gets the CM_PROGRAM_PARAMETER_CHECK return code.

SEE ALSO

Section 4.3.2 on page 72 and Section 4.3.4 on page 76 show example program flows where the Prepare_To_Receive call is used.

Set_Confirmation_Urgency (CMSCU) on page 263 tells how to request an immediate response to the Prepare_To_Receive call. *Set_Prepare_To_Receive_Type (CMSPTR)* on page 293 provides more information on the *prepare_to_receive_type* characteristic.

Set_Sync_Level (CMSSL) on page 311 discusses the *sync_level* characteristic and its possible values.

NAME

Receive (CMRCV) — receive data.

SYNOPSIS

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

DESCRIPTION

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.

The Receive (CMRCV) call uses the following input and output 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 the **APPLICATION USAGE** section below 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* has no meaning.

  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 or control information 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* has no meaning.

  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* has no meaning.

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

  CM_NO_STATUS_RECEIVED
  No conversation status is received by the program; data or control information may be received.

  CM_SEND_RECEIVED (half-duplex conversations only)
  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 (half-duplex conversations only)
  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, Deallocate with *deallocate_type* set to CM_DEALLOCATE_ABEND, or Cancel_Conversation.

  CM_CONFIRM_SEND_RECEIVED (half-duplex conversations only)
  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, Deallocate with *deallocate_type* set to CM_DEALLOCATE_ABEND, or

Cancel_Conversation. 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, Deallocate with *deallocate_type* set to CM_DEALLOCATE_ABEND, or Cancel_Conversation. 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 or CM_SYNC_POINT_NO_CONFIRM, and with the conversation included in a transaction, 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 or a Prepare call. For the exact conditions for receipt of this *status_received* value, refer to Table 3-7 on page 53 (half-duplex) or Table 3-8 on page 54 (full-duplex). 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 (for half-duplex conversations only) or a resource recovery backout call.

CM_TAKE_COMMIT_SEND (for half-duplex conversations only)
: 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 or a Prepare call. For the exact conditions for receipt of this *status_received* value, refer to Table 3-7 on page 53 (half-duplex) or Table 3-8 on page 54 (full-duplex). 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 or CM_SYNC_POINT_NO_CONFIRM, or has issued a Deferred_Deallocate call, and then issued a resource recovery commit call ore a Prepare call. For the exact conditions for receipt of this *status_received* value, refer to Table 3-7 on page 53 (half-duplex) or Table 3-8 on page 54 (full-duplex). 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 for a half-duplex conversation, 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.

CM_PREPARE_OK
: By issuing a Prepare (CMPREP) call, the local program requested that the remote program prepare its resources for commitment, and the remote program has done so, by issuing a commit call. The subtree is now ready to commit its resources.

For a conversation with *sync_level* set to either CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and allocated using an OSI TP CRM, the *status_received* variable can also be set to one of the following values if the conversation is included in a transaction:

CM_TAKE_COMMIT_DATA_OK
: The remote program issued a Prepare call. For the exact conditions for receipt of this *status_received* value, refer to Table 3-7 on page 53 (half-duplex) or Table 3-8 on page 54 (full-duplex). The local program should issue a commit call in order to commit all protected resources throughout the transaction. The program is allowed to send data before issuing the commit call. When appropriate, the local program may respond by issuing a call other than commit, such as Send_Error (for half-duplex conversations only) or a resource recovery backout call.

CM_TAKE_COMMIT_SEND_DATA_OK (half-duplex conversations only)
: The remote program issued a Prepare call. For the exact conditions for receipt of this *status_received* value, refer to Table 3-7 on page 53 (half-duplex) or Table 3-8 on page 54 (full-duplex). The local program should issue a commit call in order to commit all protected resources throughout the transaction. The program is allowed to send data before issuing the commit call. 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_DEALLOC_DATA_OK
: The remote program issued a Prepare call. For the exact conditions for receipt of this *status_received* value, refer to Table 3-7 on page 53 (half-duplex) or Table 3-8 on page 54 (full-duplex). The local program should issue a commit call in order to commit all protected resources throughout the transaction. The program is allowed to send data before issuing the commit call. The local program may respond by issuing a call other than commit when appropriate, such as Send_Error for a half-duplex conversation, 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.

CM_JOIN_TRANSACTION (unchained transactions only)
: The remote program has requested that the local program join into its current transaction. If the local program has called Set_Join_Transaction with CM_JOIN_EXPLICIT, the local program should issue a *tx_begin*( ) call to the TX (Transaction Demarcation) interface to join the superior's transaction as soon as any local work that is not to be included in the remote program's transaction has been completed. If the local program has called Set_Join_Transaction with CM_JOIN_IMPLICIT, the local program has already joined the transaction, and *tx_begin*( ) must not be called.

- *control_information_received* (output)

Specifies the variable containing an indication of whether or not control information has been received.

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

CM_NO_CONTROL_INFO_RECEIVED
: Indicates that no control information was received.

CM_REQ_TO_SEND_RECEIVED (half-duplex conversations only)
: 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 227 for further discussion of the local program's possible responses.

CM_ALLOCATE_CONFIRMED (OSI TP CRM only)
The local program received confirmation of the remote program's acceptance of the conversation.

CM_ALLOCATE_CONFIRMED_WITH_DATA (OSI TP CRM only)
The local program received confirmation of the remote program's acceptance of the conversation. The local program may now issue an Extract_Initialization_Data (CMEID) call to receive the initialization data.

CM_ALLOCATE_REJECTED_WITH_DATA (OSI TP CRM only)
The remote program rejected the conversation. The local program may now issue an Extract_Initialization_Data (CMEID) call to receive the initialization data. This value will be returned with a return code of CM_OK. The program will receive a CM_DEALLOCATED_ABEND return code on a later call on the conversation.

CM_EXPEDITED_DATA_AVAILABLE (LU 6.2 CRM only)
Expedited data is available to be received.

CM_RTS_RCVD_AND_EXP_DATA_AVAIL (half-duplex and LU 6.2 CRM only)
The local program received a request-to-send notification from the remote program and expedited data is available to be received.

Notes:

1. If *return_code* is set to CM_PROGRAM_PARAMETER_CHECK or CM_PROGRAM_STATE_CHECK, the value contained in *control_information_received* has no meaning.
2. When more than one piece of control information is available to be returned to the program, it will be returned in the following order:
   - CM_ALLOCATE_CONFIRMED, CM_ALLOCATE_CONFIRMED_WITH_DATA or CM_ALLOCATE_REJECTED_WITH_DATA
   - CM_RTS_RCVD_AND_EXP_DATA_AVAIL
   - CM_REQ_TO_SEND_RECEIVED
   - CM_EXPEDITED_DATA_AVAILABLE
   - CM_NO_CONTROL_INFO_RECEIVED.

- *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.

Half-duplex Conversations

The following return codes apply to half-duplex conversations:

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_SYNC_LVL_NOT_SUPPORTED_SYS

CM_SEND_RCV_MODE_NOT_SUPPORTED

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 or CM_SYNC_POINT_NO_CONFIRM and the conversation is included in a transaction:

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

CM_INCLUDE_PARTNER_REJECT_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_SYNC_LVL_NOT_SUPPORTED_SYS

CM_SEND_RCV_MODE_NOT_SUPPORTED

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 or CM_SYNC_POINT_NO_CONFIRM and the conversation is included in a transaction:

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

CM_INCLUDE_PARTNER_REJECT_BO

If *receive_type* is set to CM_RECEIVE_AND_WAIT or CM_RECEIVE_IMMEDIATE and this call is issued in **Receive** or **Prepared** 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_SYNC_LVL_NOT_SUPPORTED_SYS

CM_SEND_RCV_MODE_NOT_SUPPORTED

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 or CM_SYNC_POINT_NO_CONFIRM and the conversation is included in a transaction:

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

CM_INCLUDE_PARTNER_REJECT_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**, **Receive** or **Prepared** state.

— The *receive_type* is set to CM_RECEIVE_IMMEDIATE and the conversation is not in **Receive** or **Prepared** 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 or CM_SYNC_POINT_NO_CONFIRM and *transaction_control* set to

CM_CHAINED_TRANSACTIONS or *begin_transaction* set to CM_BEGIN_IMPLICIT, the program is in the **Backout-Required** condition. The Receive call is not allowed for this conversation while the program is in this condition.

— The program has received a *status_received* value of CM_JOIN_TRANSACTION and must issue a *tx_begin*( ) call to the TX (Transaction Demarcation) interface to join the transaction.

— The Receive call is the first activity on the conversation following Accept_Conversation or Accept_Incoming, *join_transaction* is set to CM_JOIN_EXPLICIT, *transaction_control* is CM_CHAINED_TRANSACTION and the program has not issued a *tx_begin*( ) call to the TX (Transaction Demarcation) interface to join the transaction.

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 the **APPLICATION USAGE** section below for additional information about determining the maximum buffer size.

Full-duplex Conversations

The following return codes apply to full-duplex conversations:

If the call is issued in **Send-Receive** or **Receive-Only** state and either the *sync_level* is CM_NONE or the *sync_level* is CM_SYNC_POINT_NO_CONFIRM and the conversation is not currently included in a transaction, *return_code* can be one of the following:

CM_OK

CM_PIP_NOT_SPECIFIED_CORRECTLY

CM_CONVERSATION_TYPE_MISMATCH

CM_SECURITY_NOT_VALID

CM_SYNC_LVL_NOT_SUPPORTED_PGM

CM_SYNC_LVL_NOT_SUPPORTED_SYS

CM_SEND_RCV_MODE_NOT_SUPPORTED

CM_TPN_NOT_RECOGNIZED

CM_TP_NOT_AVAILABLE_NO_RETRY

CM_TP_NOT_AVAILABLE_RETRY

CM_DEALLOCATED_NORMAL

CM_DEALLOCATED_ABEND

CM_DEALLOCATED_ABEND_SVC (basic conversations only)

CM_DEALLOCATED_ABEND_TIMER (basic conversations only)

CM_PROGRAM_ERROR_NO_TRUNC

CM_PROGRAM_ERROR_TRUNC (basic conversations only)

CM_PROGRAM_ERROR_PURGING (OSI TP CRM only)

CM_RESOURCE_FAILURE_RETRY

CM_RESOURCE_FAILURE_NO_RETRY

CM_SVC_ERROR_NO_TRUNC (basic conversations only)

CM_SVC_ERROR_TRUNC (basic conversations only)

CM_OPERATION_NOT_ACCEPTED

CM_OPERATION_INCOMPLETE

CM_PRODUCT_SPECIFIC_ERROR

CM_UNSUCCESSFUL
This value indicates that *receive_type* is set to CM_RECEIVE_IMMEDIATE, but there is nothing to receive.

The following values are returned only if *sync_level* is CM_SYNC_POINT_NO_CONFIRM and the state is **Send-Receive** or **Prepared** and the conversation is included in a transaction:

CM_TAKE_BACKOUT

CM_DEALLOCATED_ABEND_BO

CM_DEALLOCATED_ABEND_SCV_BO (basic conversations only)

CM_DEALLOCATED_ABEND_TIMER_BO (basic conversations only)

CM_RESOURCE_FAIL_NO_RETRY_BO

CM_RESOURCE_FAILURE_RETRY_BO

CM_CONV_DEALLOC_AFTER_SYNCPT

CM_INCLUDE_PARTNER_REJECT_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 conversation is not in **Send-Receive**, **Prepared** or **Receive-Only** state.

— For a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and *transaction_control* set to CM_CHAINED_TRANSACTIONS or *begin_transaction* set to CM_BEGIN_IMPLICIT, the program is in the **Backout-Required** condition. The Receive call is not allowed for this conversation while the program is in this condition.

— The local program has received a *status_received* value of CM_JOIN_TRANSACTION and must issue a *tx_begin*( ) call to the TX (Transaction Demarcation) interface to join the transaction.

— The Receive call is the first activity on the conversation following Accept_Conversation or Accept_Incoming, *join_transaction* is set to CM_JOIN_EXPLICIT, *transaction_control* is CM_CHAINED_TRANSACTION and the program has not issued a *tx_begin*( ) call to the TX (Transaction Demarcation) interface to join the transaction.

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 32,767. See the **APPLICATION USAGE** section below for additional information about determining the maximum buffer size.

STATE CHANGES

For half-duplex conversations, 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 either CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, 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.
- For a conversation with *sync_level* set to either CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the conversation enters **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** state when *status_received* is set to CM_TAKE_COMMIT_DATA_OK, CM_TAKE_COMMIT_SEND_DATA_OK or CM_TAKE_COMMIT_DEALLOC_DATA_OK, 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.
- No state change occurs when the call is issued in **Prepared** state, or if *status_received* indicates CM_JOIN_TRANSACTION.

For full-duplex conversations, when *return_code* indicates CM_OK:

- No state change occurs when the call is issued in **Prepared** or **Receive-Only** state, or if *status_received* indicates CM_JOIN_TRANSACTION.

- The conversation enters **Confirm-Deallocate** state when *status_received* is set to CM_CONFIRM_DEALLOC_RECEIVED.
- For a conversation with *sync_level* set to CM_SYNC_POINT_NO_CONFIRM, the conversation enters **Sync-Point** state when *status_received* is set to CM_TAKE_COMMIT or CM_TAKE_COMMIT_DATA_OK, and it enters **Sync-Point-Deallocate** state when *status_received* is set to CM_TAKE_COMMIT_DEALLOCATE or CM_TAKE_COMMIT_DEALLOC_DATA_OK.

APPLICATION USAGE

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. For a half-duplex conversation, 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 flushes its send buffer and sends all buffered information to the remote program. The local system also sends 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 202 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 Section 3.7.1 on page 26.
4. If the *return_code* indicates CM_PROGRAM_STATE_CHECK or CM_PROGRAM_PARAMETER_CHECK, the values of all other parameters on this call have no meaning.
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 for one of the following reasons:

      — *receive_type* is set to CM_RECEIVE_AND_WAIT, and the length of the record is greater than the length specified on the *requested_length* parameter.

— *receive_type* is set to CM_RECEIVE_IMMEDIATE, and either the length of the record is greater than the length specified on the *requested_length* parameter or the last portion of the data record has not arrived from the partner program.

In either case, the amount of data received is equal to the *received_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 (possibly 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:

— for a half-duplex conversation, a change to **Send**, **Send-Pending**, **Confirm**, **Confirm-Send**, **Confirm-Deallocate**, **Sync-Point**, **Sync-Point-Send**, **Sync-Point-Deallocate** or **Reset** state

— for a full-duplex conversation, a change to **Send-Only**, **Confirm-Deallocate**, **Sync-Point**, **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 program may receive data and conversation status on the same Receive call or on separate Receive calls. The program should be prepared for either case.

11. For a half-duplex conversation, 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.

12. 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:

    1. Never using a *requested_length* value greater than 32767.
    2. 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.

13. When the Receive call is processed in non-blocking mode 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.

14. When the local program has requested confirmation of the Allocate call and the first call made by the recipient program is Request_To_Send or the recipient program has issued a Send_Expedited_Data call, the CM_ALLOCATE_CONFIRMED value of the

control_information_received parameter is returned first, and one of the following values is returned at the next opportunity:

CM_RTS_RCVD_AND_EXP_DATA_AVAIL

CM_REQ_TO_SEND_RECEIVED

CM_EXPEDITED_DATA_AVAILABLE.

15. The Receive call may be issued following a successful Prepare call, without a state transition to **Receive** state in case of a half-duplex conversation, for either of these reasons:

    — to receive data when CM_PREPARE_DATA_PERMITTED is selected and the Prepare Call is issued

    — to receive a new CM_PREPARE_OK value in *status_received*.

16. For a full-duplex conversation, if *receive_type* is set to CM_RECEIVE_AND_WAIT and the conversation startup request has not been sent to the partner, then the Receive call will flush the conversation startup request to the partner.

17. For a full-duplex conversation, if the return code CM_PROGRAM_ERROR_PURGING is received, it indicates that the conversation is allocated using an OSI TP CRM and that data may have been purged. The application has to ensure that the two partners are coordinated.

18. For a full-duplex conversation, when CM_DEALLOCATED_ABEND or CM_DEALLOCATED_ABEND_BO is received, further information on the cause of the deallocation may be obtained by issuing the Extract_Secondary_Information call.

19. When *control_information_received* indicates that expedited data is available to be received, subsequent calls with this parameter continue to indicate that expedited data is available until the expedited data has been received by the program.

SEE ALSO

Section 3.2 on page 19 and *Set_Fill (CMSF)* on page 279 further discuss the use of basic conversations.

Most of the example program flows in Chapter 4 show programs using the Receive call.

Extract_Maximum_Buffer_Size (CMEMBS) on page 172 further discusses determining the maximum buffer size supported by the system.

Request_To_Send (CMRTS) on page 227 discusses how a program can place its end of the conversation into **Receive** state.

Send_Data (CMSEND) on page 230 provides more information on complete and incomplete logical records and data records.

Set_Receive_Type (CMSRT) on page 304 discusses the *receive_type* characteristic and its various values.

NAME

Receive_Expedited_Data (CMRCVX) — receive expedited data from its partner.

SYNOPSIS

```
CALL CMRCVX(conversation_ID,buffer,requested_length,received_length,
            control_information_received,expedited_receive_type,
            return_code)
```

DESCRIPTION

A program uses the Receive_Expedited_Data (CMRCVX) call to receive expedited data sent by its partner.

This call has meaning only when an LU 6.2 CRM is used for the conversation.

The Receive_Expedited_Data (CMRCVX) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *buffer* (output)

  Specifies the variable in which the program is to receive the data.

- *requested_length* (input)

  Specifies the maximum amount of data the program is to receive. This length can range from 0 to 86 bytes.

- *received_length* (output)

  When CM_OK is returned to the program, this parameter specifies the amount of data received, which is less than or equal to the buffer size specified in *requested_length*. When CM_BUFFER_TOO_SMALL is returned to the program, this parameter indicates the size of the data that is available to be received but has not been received.

- *control_information_received* (output)

  Specifies the variable containing an indication of whether or not control information has been received.

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

  CM_NO_CONTROL_INFO_RECEIVED
  : Indicates that no control information was received.

  CM_REQ_TO_SEND_RECEIVED (half-duplex conversations only)
  : 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 227 for further discussion of the local program's possible responses.

  CM_EXPEDITED_DATA_AVAILABLE
  : Expedited data is available to be received.

  CM_RTS_RCVD_AND_EXP_DATA_AVAIL (half-duplex conversations only)
  : The local program received a request-to-send notification from the remote program and expedited data is available to be received.

Notes:

1. If *return_code* is set to CM_PROGRAM_PARAMETER_CHECK or CM_PROGRAM_STATE_CHECK, the value contained in *control_information_received* has no meaning.
2. When more than one piece of control information is available to be returned to the program, they are returned in the following order:

   CM_RTS_RCVD_AND_EXP_DATA_AVAIL

   CM_REQ_TO_SEND_RECEIVED

   CM_EXPEDITED_DATA_AVAILABLE

   CM_NO_CONTROL_INFO_RECEIVED.

- *expedited_receive_type* (input)

  Specifies whether control should be returned to the program immediately or after there is expedited data available to receive.

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

  CM_RECEIVE_AND_WAIT
  : The Receive_Expedited_Data call is to wait for expedited data to arrive on the specified conversation. If expedited data is already available, the program receives it without waiting.

  CM_RECEIVE_IMMEDIATE
  : The Receive_Expedited_Data call is to receive any expedited data that is available from the specified conversation, but is not to wait for expedited data 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_CALL_NOT_SUPPORTED

  CM_OPERATION_INCOMPLETE

  CM_PROGRAM_PARAMETER_CHECK

  — The *conversation_ID* specifies an unassigned conversation identifier.

  — The *requested_length* specifies a value less than 0 or greater than 86.

  — The conversation is not using an LU 6.2 CRM.

  — The *expedited_receive_type* specifies an undefined value.

  CM_PROGRAM_STATE_CHECK
  : This value indicates that the conversation is in **Initialize** or **Initialize-Incoming** state and is not allowed to send expedited data.

  CM_CONVERSATION_ENDING
  : This value indicates that the conversation is ending due to a normal deallocation, an allocation error, a Cancel_Conversation call, a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND, or a conversation failure. Hence, no expedited data is received.

CM_EXP_DATA_NOT_SUPPORTED
: This value indicates that the remote system does not support expedited data.

CM_BUFFER_TOO_SMALL
: This value indicates that the value specified for the *requested_length* parameter is less than the amount of expedited data to be received. Therefore, no expedited data has been received.

CM_OPERATION_NOT_ACCEPTED

CM_UNSUCCESSFUL
: This value indicates that the *expedited_receive_type* parameter was set to CM_RECEIVE_IMMEDIATE and there was no expedited data available to receive.

CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause a state change.

APPLICATION USAGE

1. When expedited data is received by the CRM from the partner program, it is indicated to the local program on the next call it issues that returns the *control_information_received* parameter. When the program uses multiple threads or queue-level non-blocking, more than one call with the *control_information_received* parameter may be executed simultaneously. The availability of expedited data will continue to be indicated until the expedited data is received by the program. However, if a request-to-send or an allocate-confirm notification has been received, this notification is given to the program in only one call that has the *control_information_received* parameter.
2. If the program issues Receive_Expedited_Data with *requested_length* set to 0 and there is data available to be received, CM_BUFFER_TOO_SMALL is returned.
3. If the program issues Receive_Expedited_Data with *requested_length* set to 0 and *expedited_receive_type* set to CM_RECEIVE_AND_WAIT, and there is no data available to be received, the call does not complete until expedited data is available to be received. CM_BUFFER_TOO_SMALL is then returned.

SEE ALSO

Send_Expedited_Data (CMSNDX) on page 250 describes the Send_Expedited_Data call.

NAME

Release_Local_TP_Name (CMRLTP) — release a name.

SYNOPSIS

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

DESCRIPTION

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

The Release_Local_TP_Name (CMRLTP) call uses the following input and output 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_CALL_NOT_SUPPORTED

  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.

APPLICATION USAGE

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 is not 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 receive the CM_PROGRAM_STATE_CHECK return code.

SEE ALSO

Specify_Local_TP_Name (CMSLTP) on page 317 describes how local names are associated with a program.

NAME

Request_To_Send (CMRTS) — notify its partner that it would like to send data.

SYNOPSIS

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

DESCRIPTION

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.

Note: The Request_To_Send call has meaning only on a half-duplex conversation.

The Request_To_Send (CMRTS) call uses the following input and output 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_PARAMETER_CHECK
  This value indicates one of the following:

  — The *conversation_ID* specifies an unassigned conversation identifier.

  — The *send_receive_mode* is CM_FULL_DUPLEX.

  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**, **Sync-Point-Deallocate** or **Prepared** state.

  — For a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the program is in the **Backout-Required** condition. The Request_To_Send call is not allowed for this conversation while the program is in this condition.

  — For a conversation using an OSI TP CRM, the Request_To_Send call is not allowed from **Send** or **Prepared** state.

  — The program has received a *status_received* value of CM_JOIN_TRANSACTION and must issue a *tx_begin*( ) call to the TX (Transaction Demarcation) interface to join the transaction.

  CM_CONVERSATION_ENDING
  This return code indicates that the local system is ending the conversation or notification has been received from the remote system that it is ending the conversation.

  CM_OPERATION_NOT_ACCEPTED

  CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause a state change.

APPLICATION USAGE

1. The remote program is informed of the arrival of a request-to-send notification by means of the *control_information_received* parameter. The *control_information_received* parameter set to CM_REQ_TO_SEND_RECEIVED or CM_RTS_RCVD_AND_EXP_DATA_AVAIL 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, or CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, but with the conversation not currently included in a transaction

   — 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, or CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, but with the conversation not currently included in a transaction

   — the Prepare_To_Receive call with *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, 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 or CM_SYNC_POINT_NO_CONFIRM, 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 *Receive (CMRCV)* on page 208 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 *control_information_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 *control_information_received* parameter. The remote system retains 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.

SEE ALSO

Section 4.3.4 on page 76 shows an example program flow using the Request_To_Send call.

Receive (CMRCV) on page 208 provides additional information on the *status_received* and *control_information_received* parameters.

NAME

Send_Data (CMSEND) — send data.

SYNOPSIS

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

DESCRIPTION

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 set up may include:

CALL CMSPTR – Set_Prepare_To_Receive_Type

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

CALL CMSDT – Set_Deallocate_Type

The Send_Data (CMSEND) call uses the following input and output 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 the **APPLICATION USAGE** section below 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 set-up characteristics are significant and retain their meaning as described.

- *control_information_received* (output)

  Specifies the variable containing an indication of whether or not control information has been received.

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

  CM_NO_CONTROL_INFO_RECEIVED
  : Indicates that no control information was received.

  CM_REQ_TO_SEND_RECEIVED (half-duplex conversations only)
  : 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 227 for further discussion of the local program's possible responses.

  CM_ALLOCATE_CONFIRMED (OSI TP CRM only)
  : The local program received confirmation of the remote program's acceptance of the conversation.

  CM_ALLOCATE_CONFIRMED_WITH_DATA (OSI TP CRM only)
  : The local program received confirmation of the remote program's acceptance of the conversation. The local program may now issue an Extract_Initialization_Data (CMEID) call to receive the initialization data.

  CM_ALLOCATE_REJECTED_WITH_DATA (OSI TP CRM only)
  : The remote program rejected the conversation. The local program may now issue an Extract_Initialization_Data (CMEID) call to receive the initialization data. This value will be returned with a return code of CM_OK. The program will receive a CM_DEALLOCATED_ABEND return code on a later call on the conversation.

  CM_EXPEDITED_DATA_AVAILABLE (LU 6.2 CRM only)
  : Expedited data is available to be received.

  CM_RTS_RCVD_AND_EXP_DATA_AVAIL (half-duplex and LU 6.2 CRM only)
  : The local program received a request-to-send notification from the remote program and expedited data is available to be received.

Notes:

1. If *return_code* is set to CM_PROGRAM_PARAMETER_CHECK or CM_PROGRAM_STATE_CHECK, the value contained in *control_information_received* has no meaning.
2. When more than one piece of control information is available to be returned to the program, they are returned in the following order:
   - CM_ALLOCATE_CONFIRMED, CM_ALLOCATE_CONFIRMED_WITH_DATA or CM_ALLOCATE_REJECTED_WITH_DATA
   - CM_RTS_RCVD_AND_EXP_DATA_AVAIL
   - CM_REQ_TO_SEND_RECEIVED
   - CM_EXPEDITED_DATA_AVAILABLE
   - CM_NO_CONTROL_INFO_RECEIVED.

- *return_code* (output)

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

  Half-duplex Conversations

  The following return codes apply to half-duplex conversations:

  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_SYNC_LVL_NOT_SUPPORTED_SYS

  CM_SEND_RCV_MODE_NOT_SUPPORTED

  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**, **Send-Pending**, **Sync-Point**, **Sync-Point-Send** or **Sync-Point-Deallocate** state.

— The conversation is in **Sync-Point**, **Sync-Point-Send** or **Sync-Point-Deallocate** state, and the program receives a take-commit notification not ending in *_DATA_OK.

— 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 or CM_SYNC_POINT_NO_CONFIRM and *transaction_control* set to CM_CHAINED_TRANSACTIONS or *begin_transaction* set to CM_BEGIN_IMPLICIT, the program is in the **Backout-Required** condition. The Send_Data call is not allowed for this conversation while the program is in this condition.

— The conversation is in **Sync-Point**, **Sync-Point-Send** or **Sync-Point-Deallocate** state, and *send_type* is set to CM_SEND_AND_CONFIRM or CM_SEND_AND_PREP_TO_RECEIVE.

— The conversation is in **Sync-Point**, **Sync-Point-Send** or **Sync-Point-Deallocate** state, the *send_type* is set to CM_SEND_AND_DEALLOCATE, and the *deallocate_type* is not set to CM_DEALLOCATE_ABEND.

— The *send_type* is set to CM_SEND_AND_DEALLOCATE and the following conditions are also true:

  — The *deallocate_type* is set to CM_DEALLOCATE_FLUSH or CM_DEALLOCATE_CONFIRM.

  — The *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.

  — The conversation is included in a transaction.

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 the **APPLICATION USAGE** section below 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'.

— The *send_type* is set to CM_SEND_AND_PREP_TO_RECEIVE and the following conditions are also true:

  — The *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL.

  — The *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.

— The conversation is included in a transaction.

— The conversation is using an OSI TP CRM, and the program is not the superior for the conversation.

— The *send_type* is set to CM_SEND_AND_DEALLOCATE and the following conditions are also true:

— The *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL.

— The *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.

— The conversation is included in a transaction.

— The conversation is using an OSI TP CRM, and the program is not the superior for the conversation.

CM_OPERATION_NOT_ACCEPTED

CM_PRODUCT_SPECIFIC_ERROR

The following values are returned only when *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and the conversation is included in a transaction:

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

CM_INCLUDE_PARTNER_REJECT_BO.

Full-duplex Conversations

The following return codes apply to full-duplex conversations:

The *return_code* can have one of the following values:

CM_OK

CM_ALLOCATION_ERROR

CM_DEALLOCATED_ABEND

CM_DEALLOCATED_ABEND_SVC

CM_DEALLOCATED_ABEND_TIMER

CM_DEALLOC_CONFIRM_REJECT

CM_RESOURCE_FAILURE_NO_RETRY

CM_RESOURCE_FAILURE_RETRY

CM_DEALLOCATED_NORMAL (OSI TP CRM only)

CM_PROGRAM_STATE_CHECK

This value indicates one of the following:

— The conversation is not in **Send-Receive**, **Send-Only**, **Sync-Point** or **Sync-Point-Deallocate** state.

— The conversation is basic and in **Send-Receive** or **Send-Only** state, the *send_type* is set to CM_SEND_AND_DEALLOCATE, the *deallocate_type* is not set to CM_DEALLOCATE_ABEND, and the data does not end on a logical record boundary.

— The conversation is in **Sync-Point** or **Sync-Point Deallocate** state and the program received a take-commit notification not ending in *_DATA_OK.

— The conversation is in **Sync-Point** or **Sync-Point-Deallocate** state, the *send_type* is set to CM_SEND_AND_DEALLOCATE, and the *deallocate_type* is not set to CM_DEALLOCATE_ABEND.

— For a conversation with *sync_level* set to CM_SYNC_POINT_NO_CONFIRM, this return code indicates one of the following:

  — The *transaction_control* is set to CM_CHAINED_TRANSACTIONS or the *begin_transaction* set to CM_BEGIN_IMPLICIT, and the program is in the **Backout-Required** condition. The Send_Data call is not allowed for this conversation while the program is in this condition.

  — The local program has received a *status_received* value of CM_JOIN_TRANSACTION and must issue a *tx_begin*( ) call to the TX (Transaction Demarcation) interface to join the transaction.

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.

— 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'.

— The *send_type* is set to CM_SEND_AND_DEALLOCATE and the following conditions are also true:

  — The *deallocate_type* is set to CM_DEALLOCATE_FLUSH or CM_DEALLOCATE_CONFIRM.

  — The *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM.

  — The conversation in included in a transaction.

— The *send_type* is set to CM_SEND_AND_DEALLOCATE and the following conditions are also true:

  — The *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL.

  — The *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM.

  — The conversation is included in a transaction.

  — The program is not the superior for the conversation.

CM_OPERATION_NOT_ACCEPTED

CM_OPERATION_INCOMPLETE

CM_PRODUCT_SPECIFIC_ERROR.

The following values are returned only if *sync_level* is CM_SYNC_POINT_NO_CONFIRM, the state is **Send-Receive** and the conversation is currently included in a transaction:

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

CM_CONV_DEALLOC_AFTER_SYNCPT

CM_INCLUDE_PARTNER_REJECT_BO.

STATE CHANGES

For half-duplex conversations, 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_FLUSH.
  - — *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_NONE or CM_CONFIRM.
  - — *Prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, but the conversation is not currently included in a transaction.
- 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, *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and the conversation is included in a transaction.
- 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_FLUSH.
  - — *Deallocate_type* is set to CM_DEALLOCATE_CONFIRM.
  - — *Deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is set to CM_NONE or CM_CONFIRM.
  - — *Deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, but the conversation is not currently included in a transaction.

- 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, *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and the conversation is included in a transaction.
- 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.

For full-duplex conversations, when *return_code* indicates CM_OK:

- The conversation enters **Receive-Only** state when the Send_Data call is issued in **Send-Receive** state with *send_type* set to CM_SEND_AND_DEALLOCATE, and one of the following conditions is true:
  - — *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.
  - — *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, and the conversation is not currently included in a transaction.
- The conversation enters **Reset** state when the Send_Data call is issued with *send_type* set to CM_SEND_AND_DEALLOCATE and one of the following conditions is true:
  - — The call is issued in **Send-Only** state.
  - — The call is issued in **Send-Receive**, **Send-Only**, **Sync-Point** or **Sync-Point-Deallocate** state and *deallocate_type* is set to CM_DEALLOCATE_ABEND.
- The conversation enters **Defer-Deallocate** state when the Send_Data call is issued with *send_type* set to CM_SEND_AND_DEALLOCATE, *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* set to CM_SYNC_POINT_NO_CONFIRM, and the conversation included in a transaction.
- No state change occurs when Send_Data is issued in **Send-Receive** or **Send-Only** state with *send_type* set to CM_BUFFER_DATA or CM_SEND_AND_FLUSH.

APPLICATION USAGE

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 logical connection allocated for the conversation, and varies from one logical connection to another.
2. For a half-duplex conversation, when *control_information_received* indicates CM_REQ_TO_SEND_RECEIVED, or CM_RTS_RCVD_AND_EXP_DATA_AVAIL, or 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 227 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

   — Include_Partner_In_Transaction

   — Prepare

   — 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. When a Send_Data call is issued with *send_type* set to CM_SEND_AND_DEALLOCATE, *deallocate_type* set to CM_DEALLOCATE_ABEND and the conversation is included in a transaction, the program may be placed in the **Backout-Required** condition.
10. 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:

    — never using a *send_length* value greater than 32767

    — 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.
11. When *control_information_received* indicates that expedited data is available to be received, subsequent calls with this parameter will continue to indicate that expedited data is available until the expedited data has been received by the program.

SEE ALSO

Section 3.2 on page 19 provides more information on mapped and basic conversations.

Section 4.3.1 on page 71 provides a complete discussion of controls over data transmission.

All of the example program flows in Chapter 4 make use of the Send_Data call.

Extract_Maximum_Buffer_Size (CMEMBS) on page 172 further discusses determining the maximum buffer size supported by the system.

Receive (CMRCV) on page 208 provides more information on the *data_received* parameter.

Set_Send_Type (CMSST) on page 309 provides more information on the *send_type* conversation characteristic and the use of it in combination with calls to Send_Data.

The referenced SNA Programmer's Reference specification provides further discussion of basic conversations.

NAME

Send_Error (CMSERR) — notify its partner of an error that occurred during the conversation.

SYNOPSIS

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

DESCRIPTION

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**, **Send-Receive** or **Send-Only** state, Send_Error forces the system to flush its send buffer.

For a half-duplex conversation, 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.

For a full-duplex conversation, no state change occurs. The issuance of Send_Error will be reported to the partner on a Receive call.

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 (for half-duplex conversations only)
CALL CMSLD – Set_Log_Data.

The Send_Error (CMSERR) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *control_information_received* (output)

  Specifies the variable containing an indication of whether or not control information has been received.

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

  CM_NO_CONTROL_INFO_RECEIVED
  : Indicates that no control information was received.

  CM_REQ_TO_SEND_RECEIVED (half-duplex conversations only)
  : 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 227 for further discussion of the local program's possible responses.

  CM_ALLOCATE_CONFIRMED (OSI TP CRM only)
  : The local program received confirmation of the remote program's acceptance of the conversation.

  CM_ALLOCATE_CONFIRMED_WITH_DATA (OSI TP CRM only)
  : The local program received confirmation of the remote program's acceptance of the conversation. The local program may now issue an Extract_Initialization_Data (CMEID) call to receive the initialization data.

  CM_ALLOCATE_REJECTED_WITH_DATA (OSI TP CRM only)
  : The remote program rejected the conversation. The local program may now issue an Extract_Initialization_Data (CMEID) call to receive the initialization data. This value will be returned with a return code of CM_OK. The program will receive a CM_DEALLOCATED_ABEND return code on a later call on the conversation.

CM_EXPEDITED_DATA_AVAILABLE (LU 6.2 CRM only)
Expedited data is available to be received.

CM_RTS_RCVD_AND_EXP_DATA_AVAIL (half-duplex and LU 6.2 CRM only)
The local program received a request-to-send notification from the remote program and expedited data is available to be received.

Notes:

1. If *return_code* is set to CM_PROGRAM_PARAMETER_CHECK or CM_PROGRAM_STATE_CHECK, the value contained in *control_information_received* has no meaning.
2. When more than one piece of control information is available to be returned to the program, they are returned in the following order:
   - CM_ALLOCATE_CONFIRMED, CM_ALLOCATE_CONFIRMED_WITH_DATA or CM_ALLOCATE_REJECTED_WITH_DATA
   - CM_RTS_RCVD_AND_EXP_DATA_AVAIL
   - CM_REQ_TO_SEND_RECEIVED
   - CM_EXPEDITED_DATA_AVAILABLE
   - CM_NO_CONTROL_INFO_RECEIVED.

- *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.

  Half-duplex Conversations

  The following return codes apply to half-duplex conversations:

  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_SYNC_LVL_NOT_SUPPORTED_SYS

  CM_SEND_RCV_MODE_NOT_SUPPORTED

  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 or CM_SYNC_POINT_NO_CONFIRM and the conversation is included in a transaction:

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 or CM_SYNC_POINT_NO_CONFIRM and *transaction_control* set to CM_CHAINED_TRANSACTIONS or *begin_transaction* set to CM_BEGIN_IMPLICIT, the program is in the **Backout-Required** condition. The Send_Error call is not allowed for this conversation while the program is in this condition.

CM_RESOURCE_FAIL_NO_RETRY_BO

CM_RESOURCE_FAILURE_RETRY_BO

CM_INCLUDE_PARTNER_REJECT_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_SYNC_LVL_NOT_SUPPORTED_SYS

CM_SEND_RCV_MODE_NOT_SUPPORTED

CM_TPN_NOT_RECOGNIZED

CM_TP_NOT_AVAILABLE_NO_RETRY

CM_TP_NOT_AVAILABLE_RETRY

CM_DEALLOCATED_ABEND

CM_PROGRAM_ERROR_PURGING

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 or CM_SYNC_POINT_NO_CONFIRM and the conversation is included in a transaction:

CM_TAKE_BACKOUT

CM_DEALLOCATED_ABEND_BO

CM_DEALLOCATED_NORMAL_BO

CM_PROGRAM_STATE_CHECK
This return code indicates one of the following:

— For a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and *transaction_control* set to CM_CHAINED_TRANSACTIONS or *begin_transaction* set to CM_BEGIN_IMPLICIT, the program is in the **Backout-Required** condition. The Send_Error call is not allowed for this conversation while the program is in this condition.

— The local program has received a *status_received* value of CM_JOIN_TRANSACTION and must issue a *tx_begin*( ) call to the TX (Transaction Demarcation) interface to join the transaction.

— The Send_Error call is the first activity on the conversation following Accept_Conversation or Accept_Incoming, *join_transaction* is set to CM_JOIN_EXPLICIT, *transaction_control* is CM_CHAINED_TRANSACTION and the program has not issued a *tx_begin*( ) call to the TX (Transaction Demarcation) interface to join the transaction.

CM_RESOURCE_FAIL_NO_RETRY_BO

CM_RESOURCE_FAILURE_RETRY_BO

CM_INCLUDE_PARTNER_REJECT_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

The following values are returned only when *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM:

CM_TAKE_BACKOUT

CM_PROGRAM_STATE_CHECK
This return code indicates that for a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and *transaction_control* set to CM_CHAINED_TRANSACTIONS or *begin_transaction* set to CM_BEGIN_IMPLICIT, the program is in the **Backout-Required** condition. The Send_Error call is not allowed for this conversation while the program is in this condition.

CM_RESOURCE_FAIL_NO_RETRY_BO

CM_RESOURCE_FAILURE_RETRY_BO

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

CM_OK

CM_OPERATION_INCOMPLETE

CM_PROGRAM_PARAMETER_CHECK
The *conversation_ID* specifies an unassigned conversation identifier.

CM_PROGRAM_STATE_CHECK
This return code indicates that for a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and *transaction_control* set to CM_CHAINED_TRANSACTIONS or *begin_transaction* set to CM_BEGIN_IMPLICIT, the program is in the **Backout-Required** condition. The Send_Error call is not allowed for this conversation while the program is in this condition.

CM_RESOURCE_FAILURE_NO_RETRY

CM_RESOURCE_FAILURE_RETRY

CM_RESOURCE_FAIL_NO_RETRY_BO
This value is returned only when *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.

CM_RESOURCE_FAILURE_RETRY_BO
This value is returned only when *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.

CM_TAKE_BACKOUT

CM_OPERATION_NOT_ACCEPTED

CM_PRODUCT_SPECIFIC_ERROR.

Otherwise, the conversation is in **Reset**, **Initialize**, **Defer-Receive**, **Defer-Deallocate**, **Initialize-Incoming** or **Prepared** 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.

Full-duplex Conversations

The following return codes apply to full-duplex conversations:

CM_OK

CM_ALLOCATION_ERROR

CM_DEALLOCATED_ABEND

CM_DEALLOCATED_ABEND_SVC

CM_DEALLOCATED_ABEND_TIMER

CM_RESOURCE_FAILURE_NO_RETRY

CM_RESOURCE_FAILURE_RETRY

CM_DEALLOCATED_NORMAL (OSI TP CRM only)

CM_PROGRAM_STATE_CHECK
This value indicates one of the following:

— The conversation is not in **Send-Receive**, **Send-Only** or **Confirm-Deallocate** state.

— The local program has received a *status_received* value of CM_JOIN_TRANSACTION and must issue a *tx_begin*( ) call to the TX (Transaction Demarcation) interface to join the transaction.

— The Send_Error call is the first activity on the conversation following Accept_Conversation or Accept_Incoming, *join_transaction* is set to CM_JOIN_EXPLICIT, *transaction_control* is CM_CHAINED_TRANSACTION and the program has not issued a *tx_begin*( ) call to the TX (Transaction Demarcation) interface to join the transaction.

— For a conversation with *sync_level* set to CM_SYNC_POINT_NO_CONFIRM and *transaction_control* set to CM_CHAINED_TRANSACTIONS or *begin_transaction* set to CM_BEGIN_IMPLICIT, the program is in the **Backout-Required** condition. The Send_Error call is not allowed for this conversation while the program is in this condition.

CM_PROGRAM_PARAMETER_CHECK
This value indicates the *conversation_ID* specifies an unassigned conversation identifier.

CM_OPERATION_NOT_ACCEPTED

CM_OPERATION_INCOMPLETE

CM_PRODUCT_SPECIFIC_ERROR.

The following values are returned only if *sync_level* is CM_SYNC_POINT_NO_CONFIRM, the state is **Send-Receive**, and the conversation is included in a transaction.

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

CM_CONV_DEALLOC_AFTER_SYNCPT

CM_INCLUDE_PARTNER_REJECT_BO.

STATE CHANGES

For half-duplex conversations, 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 or CM_SYNC_POINT_NO_CONFIRM and which is included in a transaction, 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.

For full-duplex conversations, when *return_code* indicates CM_OK, the conversation enters **Send-Receive** state when this call is issued in **Confirm-Deallocate** state.

APPLICATION USAGE

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 logical connection allocated for the conversation, and varies from one logical connection 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. For a half-duplex conversation using an LU 6.2 CRM, 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 230). 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, and 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, and no purging has occurred.

5. If the conversation is using an OSI TP CRM, the remote program receives CM_PROGRAM_ERROR_PURGING regardless of the conversation state.

6. When a half-duplex conversation is using an LU 6.2 CRM and 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 *control_information_received* parameter.

7. 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.

8. 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.

   The *log_data* is not sent on the Send_Error call when an OSI TP CRM is being used for the conversation. Instead, it is ignored.

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

9. The *error_direction* characteristic is significant only when a half-duplex conversation is using an LU 6.2 CRM and Send_Error is issued 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.

10. When *control_information_received* indicates that expedited data is available, subsequent calls with this parameter will continue to indicate that expedited data is available until the expedited data has been received by the program.

11. For full-duplex conversations, the issuance of Send_Error is reported on the remote program's Receive call as one of the following return codes:

    CM_PROGRAM_ERROR_NO_TRUNC (basic and mapped conversations using an LU 6.2 CRM)

CM_PROGRAM_ERROR_TRUNC (basic conversations using an LU 6.2 CRM)

CM_PROGRAM_ERROR_PURGING (conversations using an OSI TP CRM)

No data is purged, unless the conversation is using an OSI TP CRM, in which case, the program should expect purging. The partner program may expect the CM_PROGRAM_ERROR_PURGING return code if the conversation is allocated using an OSI TP CRM.

12. Send_Error does not complete successfully if an error that causes the conversation to terminate has occurred or the remote program has issued a Deallocate with *deallocate_type* set to CM_DEALLOCATE_ABEND, or CM_DEALLOCATE_FLUSH and the conversation has been allocated using an OSI TP CRM.

    For a conversation which is not included in a transaction, a CM_DEALLOCATED_ABEND_*, CM_ALLOCATION_ERROR, CM_RESOURCE_FAILURE_*_RETRY or CM_DEALLOCATED_NORMAL return code is returned. When one of the above return codes is returned and the conversation is in **Send-Receive** state, the program can terminate the conversation by issuing Receives until it gets one of the above return codes taking it to **Reset** state, or by issuing Cancel_Conversation, Deallocate with *deallocate_type* set to CM_DEALLOCATE_ABEND, or Cancel_Conversation.

    For a conversation which is included in a transaction, CM_DEALLOCATED_ABEND_*_BO, CM_ALLOCATION_ERROR, CM_RESOURCE_FAILURE_RETRY_BO or CM_RESOURCE_FAIL_NO_RETRY_BO is returned. If CM_ALLOCATION_ERROR is returned, the program behaves as though it were not in transaction, otherwise it is in **Backout-Required** condition and in **Reset** state.

SEE ALSO

Section 4.3.5 on page 78 and Section 4.3.6 on page 80 provide example program flows using Send_Error and the **Send-Pending** state; *Set_Error_Direction (CMSED)* on page 277 provides further information on the *error_direction* characteristic.

The **APPLICATION USAGE** section of *Request_To_Send (CMRTS)* on page 227 provide more information on how a conversation enters **Receive** state.

Send_Data (CMSEND) on page 230 discusses basic conversations and logical records.

Set_Log_Data (CMSLD) on page 285 provides a description of the *log_data* characteristic.

NAME

Send_Expedited_Data (CMSNDX) — send expedited data to its partner.

SYNOPSIS

```
CALL CMSNDX(conversation_ID,buffer,send_length,
            control_information_received,return_code)
```

DESCRIPTION

A program uses the Send_Expedited_Data (CMSNDX) call to send expedited data to its partner.

This call has meaning only when an LU 6.2 CRM is used for the conversation.

The Send_Expedited_Data (CMSNDX) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *buffer* (input)

  Specifies the variable containing the data to be sent.

- *send_length* (input)

  Specifies the length of the data to be sent. The minimum amount of data that can be sent is 1 byte; the maximum is 86 bytes.

- *control_information_received* (output)

  Specifies the variable containing an indication of whether or not control information has been received.

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

  CM_NO_CONTROL_INFO_RECEIVED
  : Indicates that no control information was received.

  CM_REQ_TO_SEND_RECEIVED (half-duplex conversations only)
  : 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 227 for further discussion of the local program's possible responses.

  CM_EXPEDITED_DATA_AVAILABLE
  : Expedited data is available to be received.

  CM_RTS_RCVD_AND_EXP_DATA_AVAIL (half-duplex conversations only)
  : The local program received a request-to-send notification from the remote program and expedited data is available to be received.

  Notes:

  1. If *return_code* is set to CM_PROGRAM_PARAMETER_CHECK or CM_PROGRAM_STATE_CHECK, the value contained in *control_information_received* has no meaning.

2. When more than one piece of control information is available to be returned to the program, it will be returned in the following order:

   CM_RTS_RCVD_AND_EXP_DATA_AVAIL
   CM_REQ_TO_SEND_RECEIVED
   CM_EXPEDITED_DATA_AVAILABLE
   CM_NO_CONTROL_INFO_RECEIVED.

- *return_code* (output)

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

  CM_OK

  CM_CALL_NOT_SUPPORTED

  CM_OPERATION_INCOMPLETE

  CM_PROGRAM_PARAMETER_CHECK

  — The *conversation_ID* specifies an unassigned conversation identifier.

  — The *send_length* specifies a value less than 1 or greater than 86.

  — The conversation is not using an LU 6.2 CRM.

  CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is in **Initialize** or **Initialize-Incoming** state and is not allowed to send expedited data.

  CM_CONVERSATION_ENDING
  This value indicates that the conversation is ending due to a normal deallocation, an allocation error, a Cancel_Conversation call, a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND, or a conversation failure. Hence, no expedited data is sent.

  CM_EXP_DATA_NOT_SUPPORTED
  This value indicates that the remote system does not support expedited data.

  CM_OPERATION_NOT_ACCEPTED

  CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause a state change.

APPLICATION USAGE

1. A program uses the Send_Expedited_Data call to send data that flows in an expedited fashion, possibly bypassing data sent using the Send_Data call.
2. The Send_Expedited_Data call should be used sparingly and should not be used for sending normal data.
3. When the remote system receives the expedited data, it retains the expedited data until it is received by the partner program using Receive_Expedited_Data.

4. Implementors should note that a *control_information_received* notification can be reported on this call (associated with the Expedited-Send queue), on the Receive_Expedited_Data call (associated with the Expedited-Receive queue), on the Send_Data call (associated with the Send queue or the Send-Receive queue), on the Receive call (associated with the Receive queue or the Send-Receive queue), and on the Test_Request_To_Send_Received call (not associated with any queue). When the program uses multiple threads or queue-level non-blocking, more than one of these calls may be executed simultaneously. An implementation should report the CM_EXPEDITED_DATA_AVAILABLE indication to the program through all available calls, until the expedited data is received. All other values of *control_information_received* should be reported only once.

SEE ALSO

Receive_Expedited_Data (CMRCVX) on page 223 describes the Receive_Expedited_Data call.

NAME

Set_AE_Qualifier (CMSAEQ) — set the *AE_qualifier* conversation characteristic.

SYNOPSIS

```
CALL CMSAEQ(conversation_ID,AE_qualifier,AE_qualifier_length,
            AE_qualifier_format,return_code)
```

DESCRIPTION

Set_AE_Qualifier (CMSAEQ) is used by a program to set the *AE_qualifier*, *AE_qualifier_length*, and *AE_qualifier_format* characteristics for a conversation. Set_AE_Qualifier overrides the current values that were originally acquired from the side information using *sym_dest_name*.

Issuing this call does not change the information in the side information. It only changes the *AE_qualifier*, the *AE_qualifier_length*, and the *AE_qualifier_format* characteristics for this conversation.

Notes:

1. A program cannot issue Set_AE_Qualifier after an Allocate call is issued. Only the program that initiated the conversation (issued the Initialize_Conversation call) can issue Set_AE_Qualifier.
2. The *AE_qualifier* characteristic is used only by an OSI TP CRM.

The Set_AE_Qualifier (CMSAEQ) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *AE_qualifier* (input)

  Specifies the application-entity-qualifier that distinguishes the application-entity at the application-process where the remote program is located.

- *AE_qualifier_length* (input)

  Specifies the length of *AE_qualifier*. The length can be from 1 to 1024 bytes.

- *AE_qualifier_format* (input)

  Specifies the format of *AE_qualifier*. The *AE_qualifier_format* variable can have one of the following values:

  CM_DN
  : Specifies that the *AE_qualifier* is a distinguished name.

  CM_INT_DIGITS
  : Specifies that the *AE_qualifier* is an integer represented as a sequence of decimal digits.

- *return_code* (output)

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

  CM_OK

  CM_CALL_NOT_SUPPORTED

CM_PROGRAM_PARAMETER_CHECK
This value indicates one of the following:

— The *conversation_ID* specifies an unassigned conversation identifier.

— The *AE_qualifier_length* is set to a value less than 1 or greater than 1024.

— The *AE_qualifier_format* specifies an undefined value.

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

CM_OPERATION_NOT_ACCEPTED

CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause any state changes.

APPLICATION USAGE

1. Specify *AE_qualifier* using the local system's native encoding. CPI Communications automatically converts the *AE_qualifier* from the native encoding where necessary.
2. If a *return_code* other than CM_OK is returned on the call, the *AE_qualifier*, the *AE_qualifier_length*, and the *AE_qualifier_format* conversation characteristics remain unchanged.
3. The *AE_qualifier* may be either a distinguished name or an integer. Distinguished names may have any format and syntax that can be recognized by the local system. Integers are represented as a series of digits.

SEE ALSO

Section 3.5.2 on page 22 and note 4 of Table A-3 on page 341 provide further discussion of the *AE_qualifier* conversation characteristic.

Section 3.8.5 on page 38 provides further information on the automatic conversion of the *AE_qualifier* parameter.

NAME

Set_Allocate_Confirm (CMSAC) — set the *allocate_confirm* conversation characteristic.

SYNOPSIS

```
CALL CMSAC(conversation_ID,allocate_confirm,return_code)
```

DESCRIPTION

Set_Allocate_Confirm (CMSAC) is used by a program to set the *allocate_confirm* characteristic for a given conversation. Set_Allocate_Confirm overrides the value that was assigned when the Initialize_Conversation call was issued.

Notes:

1. A program cannot issue Set_Allocate_Confirm after an Allocate call is issued. Only the program that initiates the conversation (issues the Initialize_Conversation call) can issue Set_Allocate_Confirm.
2. The *allocate_confirm* characteristic is used only by an OSI TP CRM.

The Set_Allocate_Confirm (CMSAC) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *allocate_confirm* (input)

  Specifies whether the program is to receive notification when the remote program confirms its acceptance of the conversation. The *allocate_confirm* variable can have one of the following values:

  CM_ALLOCATE_NO_CONFIRM
  : Specifies that the program is not to receive notification when the remote program confirms its acceptance of the conversation.

  CM_ALLOCATE_CONFIRM
  : Specifies that the program is to receive notification when the remote program confirms its acceptance 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_CALL_NOT_SUPPORTED

  CM_PROGRAM_PARAMETER_CHECK
  : This value indicates one of the following:

    — The *conversation_ID* specifies an unassigned identifier.

    — The *allocate_confirm* specifies an undefined value.

    — The *allocate_confirm* specifies CM_ALLOCATE_CONFIRM, and the conversation is using an LU 6.2 CRM.

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

  CM_OPERATION_NOT_ACCEPTED

CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause any state changes.

APPLICATION USAGE

1. If a *return_code* other than CM_OK is returned on the call, the *allocate_confirm* conversation characteristic remains unchanged.
2. When the remote program confirms its acceptance of the conversation, the initiating program is notified by receiving a *control_information_received* value of CM_ALLOCATE_CONFIRMED or CM_ALLOCATE_CONFIRMED_WITH_DATA on a subsequent call.
3. After the remote program has accepted the conversation by issuing an Accept_Conversation or Accept_Incoming call, it confirms the acceptance by issuing any call other than a Cancel_Conversation, Deallocate with *deallocate_type* of CM_DEALLOCATE_ABEND or Set_* or Extract_* call. The remote program rejects the conversation by issuing a Cancel_Conversation call or a Deallocate call with *deallocate_type* of CM_DEALLOCATE_ABEND as its first operation on the conversation (other than a Set_* or Extract_* call).
4. The program that initiates the conversation (issues Initialize_Conversation) must set *allocate_confirm* to cm_allocate_confirm if it is expecting *initialization_data* to be returned from the remote program after the remote program confirms its acceptance of the conversation.

NAME

Set_AP_Title (CMSAPT) — set the *AP_title* conversation characteristic.

SYNOPSIS

```
CALL CMSAPT(conversation_ID,AP_title,AP_title_length,
            AP_title_format,return_code)
```

DESCRIPTION

Set_AP_Title (CMSAPT) is used by a program to set the *AP_title*, *AP_title_length*, and *AP_title_format* characteristics for a conversation. Set_AP_Title overrides the current values that were originally acquired from the side information using *sym_dest_name*.

Issuing this call does not change the values in the side information. It only changes the *AP_title*, the *AP_title_length*, and the *AP_title_format* characteristics for this conversation.

Notes:

1. A program cannot issue Set_AP_Title after an Allocate call is issued. Only the program that initiated the conversation (issued the Initialize_Conversation call) can issue Set_AP_Title.
2. The *AP_title* characteristic is used only by an OSI TP CRM.

The Set_AP_Title (CMSAPT) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *AP_title* (input)

  Specifies the title of the application-process where the remote program is located.

- *AP_title_length* (input)

  Specifies the length of *AP_title*. The length can be from 1 to 1024 bytes.

- *AP_title_format* (input)

  Specifies the format of *AP_title*. The *AP_title_format* variable can have one of the following values:

  CM_DN
  : Specifies that the *AP_title* is a distinguished name.

  CM_OID
  : Specifies that the *AP_title* is an object 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_CALL_NOT_SUPPORTED

CM_PROGRAM_PARAMETER_CHECK
This value indicates one of the following:

— The *conversation_ID* specifies an unassigned identifier.

— The *AP_title_length* is set to a value less than 1 or greater than 1024.

— The *AP_title_format* specifies an undefined value.

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

CM_OPERATION_NOT_ACCEPTED

CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause any state changes.

APPLICATION USAGE

1. Specify *AP_title* using the local system's native encoding. CPI Communications automatically converts the *AP_title* from the native encoding where necessary.
2. If a *return_code* other than CM_OK is returned on the call, the *AP_title*, the *AP_title_length*, and the *AP_title_format* conversation characteristics remain unchanged.
3. The *AP_title* may be either a distinguished name or an object identifier. Distinguished names can have any format and syntax that can be recognized by the local system. Object identifiers are represented as a series of digits separated by periods (for example, *n.nn.n.nnn*).

SEE ALSO

Section 3.5.2 on page 22 and note 4 of Table A-3 on page 341 provide further discussion of the *AP_title* conversation characteristic.

Section 3.8.5 on page 38 provides more information on the automatic conversion of the *AP_title* parameter.

NAME

Set_Application_Context_Name (CMSACN) — set the *application_context_name* conversation characteristic.

SYNOPSIS

```
CALL CMSACN(conversation_ID,application_context_name,
                application_context_name_length, return_code)
```

DESCRIPTION

Set_Application_Context_Name (CMSACN) is used by a program to set the *application_context_name* and *application_context_name_length* characteristics for a conversation. Set_Application_Context_Name overrides the current values that were originally acquired from the side information using *sym_dest_name*.

Issuing this call does not change the values in the side information. It only changes the *application_context_name* and *application_context_name_length* characteristics for this conversation.

Note: The *application_context_name* characteristic is used only by an OSI TP CRM.

The Set_Application_Context_Name (CMSACN) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *application_context_name* (input)

  Specifies the name of the application context to be used on the conversation. The length can be 1-256 bytes.

- *application_context_name_length* (input)

  Specifies the length of the application context name to be used on 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_CALL_NOT_SUPPORTED

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

  CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:

  — The *conversation_ID* specifies an unassigned identifier.

  — The *application_context_name_length* is set to a value less than 1 or greater than 256.

  CM_OPERATION_NOT_ACCEPTED

  CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause any state changes.

APPLICATION USAGE

1. Specify *application_context_name* using the local system's native encoding. CPI Communications automatically converts the *application_context_name* from the native encoding where necessary.
2. If a *return_code* other than CM_OK is returned on the call, the *application_context_name* and the *application_context_name_length* conversation characteristics remain unchanged.
3. The application context name is an object identifier and is represented as a series of digits separated by periods. For example, the default application context name defined by ISO for OSI TP with UDT is represented as 1.0.10026.6.2.

SEE ALSO

Section 3.5.2 on page 22 provides more information on the *application_context_name* conversation characteristic.

Section 3.8.5 on page 38 provides further information on the automatic conversion of the *application_context_name* parameter.

NAME

Set_Begin_Transaction (CMSBT) — set the *begin_transaction* conversation characteristic.

SYNOPSIS

```
CALL CMSBT(conversation_ID,begin_transaction,return_code)
```

DESCRIPTION

Set_Begin_Transaction (CMSBT) is used by a program to set the *begin_transaction* characteristic for a given conversation. Set_Begin_Transaction overrides the value that was assigned when the Initialize_Conversation call was issued.

Note: The *begin_transaction* characteristic is used only by an OSI TP CRM.

The Set_Begin_Transaction (CMSBT) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *begin_transaction* (input)

  Specifies whether the superior will explicitly or implicitly ask that the subordinate program join the transaction. The *begin_transaction* variable can have one of the following values:

  CM_BEGIN_IMPLICIT
  : Specifies that the superior implicitly asks that the subordinate join the transaction by issuing one of the following calls from **Initialize**, **Send**, **Send-Pending** or **Send-Receive** states:

    CMALLC — Allocate
    CMSERR — Send_Error
    CMPTR — Prepare_To_Receive
    CMCFM — Confirm
    CMINCL — Include_Partner_In_Transaction
    CMPREP — Prepare
    CMRCV — Receive
    CMSEND — Send

  CM_BEGIN_EXPLICIT
  : Specifies that the superior explicitly asks that the subordinate join the transaction by use of the Include_Partner_In_Transaction (CMINCL) call.

- *return_code* (output)

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

  CM_OK

  CM_CALL_NOT_SUPPORTED

  CM_PROGRAM_STATE_CHECK
  : This value indicates that the conversation is in **Initialize-Incoming** state.

  CM_PROGRAM_PARAMETER_CHECK
  : This value indicates one of the following:

    — The *conversation_ID* specifies an unassigned identifier.

    — The *begin_transaction* specifies an undefined value.

— The *transaction_control* is set to CM_CHAINED_TRANSACTIONS.

— The program is not the superior for the conversation.

CM_OPERATION_NOT_ACCEPTED
: This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.

CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause any state changes.

APPLICATION USAGE

1. If a *return_code* other than CM_OK is returned on the call, the *begin_transaction* conversation characteristic remains unchanged.
2. This call does not apply to any previous call operation still in progress.
3. The remote program receives the request to join the transaction as a *status_received* indicator of CM_JOIN_TRANSACTION on a Receive call it issues.
4. If the superior is not in transaction when it issues an Allocate, Confirm, Include_Partner_In_Transaction, Prepare, Prepare_To_Receive, Receive, Send_data or Send_Error call, the *begin_transaction* characteristic is ignored, and the subordinate is not asked to join a transaction.
5. If *begin_transaction* is set to CM_BEGIN_IMPLICIT the subordinate is asked to join the transaction only when the Allocate, Confirm, Include_Partner_In_Transaction, Prepare, Prepare_To_Receive, Receive, Send_Data or Send_Error call is returned with return_code CM_OK.
6. The call is not associated with any conversation queue. When a conversation uses queue-level non-blocking, the call does not return CM_OPERATION_NOT_ACCEPTED on the conversation.

SEE ALSO

Section 3.14.5 on page 58 discusses chained and unchained transactions.

Section 3.14.6 on page 59 discusses how a program requests the partner program to join a transaction.

NAME

Set_Confirmation_Urgency (CMSCU) — set the *confirmation_urgency* conversation characteristic.

SYNOPSIS

```
CALL CMSCU(conversation_ID,confirmation_urgency,return_code)
```

DESCRIPTION

Set_Confirmation_Urgency (CMSCU) is used by a program to set the *confirm_urgency* characteristic for a given conversation. Set_Confirmation_Urgency overrides the value that was assigned when the Initialize_Conversation, Accept_Conversation or Initialize_For_Incoming call was issued.

Note: The *confirmation_urgency* characteristic is used only for a half-duplex conversation.

The Set_Confirmation_Urgency (CMSCU) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *confirmation_urgency* (input)

  Specifies whether the response to a Prepare_To_Receive call that requests confirmation will be sent immediately. The *confirmation_urgency* variable can have one of the following values:

  CM_CONFIRMATION_NOT_URGENT
  : Specifies that the remote program's response to the confirmation request may not be sent immediately.

  CM_CONFIRMATION_URGENT
  : Specifies that the remote program's response to the confirmation request will be sent immediately.

- *return_code* (output)

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

  CM_OK

  CM_CALL_NOT_SUPPORTED

  CM_PROGRAM_PARAMETER_CHECK
  : This value indicates one of the following:

    — The *conversation_ID* specifies an unassigned identifier.

    — The *confirmation_urgency* variable specifies an undefined value.

    — The *send_receive_mode* of the conversation is CM_FULL_DUPLEX.

    — The *sync_level* is set to CM_NONE or CM_SYNC_POINT_NO_CONFIRM.

  CM_OPERATION_NOT_ACCEPTED

  CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause any state changes.

APPLICATION USAGE

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

2. When the local program issues a Prepare_To_Receive call with *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_CONFIRM or CM_PREP_TO_RECEIVE_SYNC_LEVEL with *sync_level* set to CM_CONFIRM, the remote CRM may optimize link usage by buffering the response generated by the Confirmed call until the remote program issues another call. This may increase the time the local program must wait for control to be returned to it.

   By issuing a Set_Confirmation_Urgency call with the *confirmation_urgency* parameter set to CM_CONFIRMATION_URGENT before issuing the Prepare_To_Receive call, the local program can request that the response from the Confirmed call be sent to the local program as soon as it is issued by the remote program.

SEE ALSO

See *Prepare_To_Receive (CMPTR)* on page 202 for information on requesting confirmation.

NAME

Set_Conversation_Security_Password (CMSCSP) — set the *security_password* conversation characteristic.

SYNOPSIS

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

DESCRIPTION

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 or CM_SECURITY_PROGRAM_STRONG.

The Set_Conversation_Security_Password (CMSCSP) call uses the following input and output 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_CALL_NOT_SUPPORTED

  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 or CM_SECURITY_PROGRAM_STRONG.

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.

APPLICATION USAGE

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.

SEE ALSO

Section 3.8.5 on page 38 provides further information on the automatic conversion of the *security_password* parameter.

Section 3.11 on page 47 provides further information on security.

Set_Conversation_Security_Type (CMSCST) on page 267 provides more information on the *conversation_security_type* characteristic.

Set_Conversation_Security_User_ID (CMSCSU) on page 269 provides more information on the *security_user_ID* characteristic.

NAME

Set_Conversation_Security_Type (CMSCST) — set the *conversation_security_type* conversation characteristic.

SYNOPSIS

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

DESCRIPTION

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.

The Set_Conversation_Security_Type (CMSCST) call uses the following input and output 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 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.

  CM_SECURITY_PROGRAM_STRONG
  : 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. The local CRM ensures that the *security_password* is not exposed in clear-text form on the physical network. If the local CRM cannot ensure this, then the subsequent Allocate request will fail with a *return_code* of CM_SECURITY_NOT_SUPPORTED.

- *return_code* (output)

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

  CM_OK

  CM_CALL_NOT_SUPPORTED

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

  CM_PARM_VALUE_NOT_SUPPORTED
  This value indicates that the *conversation_security_type* specifies CM_SECURITY_PROGRAM, CM_SECURITY_SAME or CM_SECURITY_PROGRAM_STRONG and the value is not supported by the local system.

  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.

APPLICATION USAGE

1. Existing OSI TP standards do not support conversation security. So for an OSI TP CRM, the only valid *conversation_security_type* value is CM_SECURITY_NONE.
2. 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.
3. A *conversation_security_type* of CM_SECURITY_PROGRAM or CM_SECURITY_PROGRAM_STRONG 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.
4. If a *return_code* other than CM_OK is returned on the call, the *conversation_security_type* is unchanged.

SEE ALSO

Section 3.11 on page 47 provides further information on security.

Set_Conversation_Security_Password (CMSCSP) on page 265 discusses setting the *security_password* characteristic.

Set_Conversation_Security_User_ID (CMSCSU) on page 269 discusses setting the *security_user_ID* characteristic.

NAME

Set_Conversation_Security_User_ID (CMSCSU) — set the *security_user_ID* conversation characteristic.

SYNOPSIS

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

DESCRIPTION

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 or CM_SECURITY_PROGRAM_STRONG.

The Set_Conversation_Security_User_ID (CMSCSU) call uses the following input and output 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_user_ID_length* characteristic is set to zero (effectively setting the *security_user_ID* characteristic to the null string), and the *security_user_ID* 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_CALL_NOT_SUPPORTED

  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 or CM_SECURITY_PROGRAM_STRONG.

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.

APPLICATION USAGE

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.

SEE ALSO

Section 3.8.5 on page 38 provides further information on the automatic conversion of the *security_user_ID* parameter.

Section 3.11 on page 47 provides further information on security.

Set_Conversation_Security_Password (CMSCSP) on page 265 provides more information on the *security_password* characteristic.

Set_Conversation_Security_Type (CMSCST) on page 267 provides more information on the *conversation_security_type* characteristic.

NAME

Set _Conversation_Type (CMSCT) — set the *conversation_type* conversation characteristic.

SYNOPSIS

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

DESCRIPTION

Set_Conversation_Type (CMSCT) is used by a program to set the *conversation_type* characteristic for a given conversation. It overrides the value that was assigned when the Initialize_Conversation or Initialize_For_Incoming call was issued.

Notes:

1. 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.
2. When using an LU 6.2 CRM, only the program that initiates the conversation (using the Initialize_Conversation call) can issue the Set_Conversation_Type call, and it must be issued before the Allocate is issued. When using an OSI TP CRM, because the mapped/basic indication is not carried by the protocol, the recipient of the conversation request must use Initialize_For_Incoming and Set_Conversation_Type to override the *conversation_type* default of CM_MAPPED_CONVERSATION. The recipient must issue these calls before the Accept_Incoming call is issued.

The Set_Conversation_Type (CMSCT) call uses the following input and output 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** or **Initialize-Incoming** 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.

APPLICATION USAGE

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.

SEE ALSO

Section 3.2 on page 19 and the **APPLICATION USAGE** section of *Send_Data (CMSEND)* on page 230 provide more information on the differences between mapped and basic conversations.

NAME

Set_Deallocate_Type (CMSDT) — set the *deallocate_type* conversation characteristic.

SYNOPSIS

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

DESCRIPTION

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.

The Set_Deallocate_Type (CMSDT) call uses the following input and output 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, or if *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM but the conversation is not currently included in a transaction, execute the function of the Flush call and deallocate the conversation normally.

    — For half-duplex conversations, if *sync_level* is set to CM_CONFIRM, or if *sync_level* is set to CM_SYNC_POINT but the conversation is not currently included in a transaction, 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 or CM_SYNC_POINT_NO_CONFIRM and the conversation is included in a transaction, 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 147 for more information about deallocating conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.

  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
For half-duplex conversations, 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.

For full-duplex conversations, execute the function of the Flush call when the conversation is in **Send-Receive** or **Send-Only** state and deallocate the conversation abnormally. Data purging can occur when the conversation is in **Send-Receive** or **Receive-Only** state. If the conversation is basic, logical-record truncation can occur when the conversation is in **Send-Receive** or **Send-Only** 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* specifies an undefined value.

— The *deallocate_type* is set to CM_DEALLOCATE_FLUSH, *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and *transaction_control* is set to CM_CHAINED_TRANSACTIONS.

— The conversation is using an LU 6.2 CRM, *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, and *sync_level* is set to CM_NONE, CM_SYNC_POINT, or CM_SYNC_POINT_NO_CONFIRM.

— The conversation is using an OSI TP CRM, *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and *transaction_control* is set to CM_CHAINED_TRANSACTIONS.

CM_PROGRAM_STATE_CHECK
The conversation is in **Initialize-Incoming** state.

CM_OPERATION_NOT_ACCEPTED

CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause a state change.

APPLICATION USAGE

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.

— For half-duplex conversations:

— If *sync_level* is set to CM_NONE, or if *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM but the conversation is not currently included in a transaction, the conversation is unconditionally deallocated.

— If *sync_level* is set to CM_CONFIRM, or if *sync_level* is set to CM_SYNC_POINT but the conversation is not currently included in a transaction, 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 or CM_SYNC_POINT_NO_CONFIRM and the conversation is included in a transaction, the deallocation is deferred 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.

— For full-duplex conversations:

— If *sync_level* is set to CM_NONE, or if *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM but the conversation is not currently included in a transaction, and the Deallocate call is issued in **Send-Receive** state, the program can no longer issue calls associated with the Send queue, but it can continue to issue calls associated with the other conversation queues. If the Deallocate call is issued in **Send-Only** state, the conversation is deallocated.

— If *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM and the conversation is included in a transaction, the deallocation is deferred 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 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. If the conversation is using an OSI TP CRM, it can also be used for conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM if the conversation is not currently included in a transaction. 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.

   For a half-duplex conversation, the conversation is deallocated. For a full-duplex conversation, the program can no longer issue calls associated with the Send queue, and the conversation is deallocated if the Deallocate call is issued in **Send-Only** state. This *deallocate_type* value can be used for conversations with *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, this *deallocate_type* value is used when the program detects an error condition that prevents further useful communication (communication 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.

SEE ALSO

Deallocate (CMDEAL) on page 147 provides further discussion on the use of the *deallocate_type* characteristic in the deallocation of a conversation.

Set_Sync_Level (CMSSL) on page 311 provides information on how the *sync_level* characteristic is used in combination with the *deallocate_type* characteristic in the deallocation of a conversation.

NAME

Set_Error_Direction (CMSED) — set the *error_direction* conversation characteristic.

SYNOPSIS

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

DESCRIPTION

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.

Note: The *error_direction* characteristic is used by an LU 6.2 CRM and only for a half-duplex conversation.

The Set_Error_Direction (CMSED) call uses the following input and output 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_CALL_NOT_SUPPORTED

  CM_PROGRAM_PARAMETER_CHECK
  : This value indicates one of the following:

    — The *conversation_ID* specifies an unassigned conversation identifier.

    — The *error_direction* specifies CM_SEND_ERROR and the conversation is using an OSI TP CRM.

    — The *error_direction* specifies an undefined value.

    — The *send_receive_mode* of the conversation is CM_FULL_DUPLEX.

  CM_OPERATION_NOT_ACCEPTED

CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause a state change.

APPLICATION USAGE

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.

SEE ALSO

Section 4.3.6 on page 80 provides an example program using Set_Error_Direction.

Send_Error (CMSERR) on page 240 provides more information on reporting errors.

Section D.3.1 on page 481 provides more information on the *error_direction* characteristic.

NAME

Set_Fill (CMSF) — set the *fill* conversation characteristic.

SYNOPSIS

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

DESCRIPTION

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 when the Initialize_Conversation, Accept_Conversation or Initialize_For_Incoming call was issued.

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

The Set_Fill (CMSF) call uses the following input and output 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* specifies an undefined value.

  CM_OPERATION_NOT_ACCEPTED

  CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause a state change.

APPLICATION USAGE

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.

SEE ALSO

Receive (CMRCV) on page 208 provides more information on how the *fill* characteristic is used for basic conversations.

NAME

Set_Initialization_Data (CMSID) — set the *initialization_data* conversation characteristic.

SYNOPSIS

```
CALL CMSID(conversation_ID,initialization_data,
                initialization_data_length,return_code)
```

DESCRIPTION

Set_Initialization_Data (CMSID) is used by a program to set the *initialization_data* and *initialization_data_length* conversation characteristics to be sent to the remote program for a given conversation. Set_Initialization_Data overrides the values that were assigned when the Initialize_Conversation, Accept_Conversation, Initialize_For_Incoming or Accept_Incoming call was issued.

The Set_Initialization_Data (CMSID) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *initialization_data* (input)

  Specifies the initialization data that is to be passed to the remote program during conversation startup.

- *initialization_data_length* (input)

  Specifies the length of the initialization data. The length can be from 0 to 10000 bytes. If zero, the *initialization_data* parameter 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_CALL_NOT_SUPPORTED

  CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:

  — The conversation is not in **Initialize** state, **Receive** state (for a half-duplex conversation), or **Send-Receive** state (for a full-duplex conversation).

  CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:

  — The *conversation_ID* specifies an unassigned identifier.

  — The *initialization_data_length* specifies a value greater than 10000 or less than zero.

  CM_OPERATION_NOT_ACCEPTED

  CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause any state changes.

APPLICATION USAGE

1. Initialization data is data that the program initiating a conversation (using the Initialize_Conversation call) can choose to send to the remote program. The initiator issues a Set_Initialization_Data call between the Initialize_Conversation and Allocate calls

to specify the initialization data to be sent by the Allocate call. Following the successful completion of either an Accept_Conversation or Accept_Incoming call, the remote program issues an Extract_Initialization_Data call to extract the initialization data.

2. When the conversation is allocated using an OSI TP CRM, the remote program may issue a Set_Initialization_Data call to identify data that is to be sent to the initiating program on its next call following its acceptance of the conversation. To avoid overwriting initialization data from the initiating program, the recipient must issue the Extract_Initialization_Data call to extract the incoming initialization data before issuing the Set_Initialization_Data call. The Set_Initialization_Data must be issued before any other call on the conversation, except Set_* or Extract_* calls. To extract the data from the remote program, the initiating program issues the Extract_Initialization_Data call after receiving a *control_information_received* value of CM_ALLOCATE_CONFIRMED_WITH_DATA or CM_ALLOCATE_REJECTED_WITH_DATA.

3. If a *return_code* other than CM_OK is returned on the call, the *initialization_data* and *initialization_data_length* conversation characteristics remain unchanged.

SEE ALSO

Extract_Initialization_Data (CMEID) on page 170 describes the Extract_Initialization_Data call.

NAME

Set_Join_Transaction (CMSJT) — set the *join_transaction* characteristic.

SYNOPSIS

```
CALL CMSJT(conversation_ID,join_transaction,return_code)
```

DESCRIPTION

Set_Join_Transaction (CMSJT) is used by a program to set the *join_transaction* characteristic for a given conversation. Set_Join_Transaction overrides the value that was assigned when the Accept_Conversation or Accept_Incoming call was issued.

Note: The *join_transaction* characteristic is only meaningful in conjunction with the TX (Transaction Demarcation) interface.

The Set_Join_Transaction (CMSJT) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *join_transaction* (input)

  Specifies whether the subordinate implicitly or explicitly joins the transaction after receiving a join transaction request from the superior. The *join_transaction* characteristic can have one of the following values:

  CM_JOIN_IMPLICIT
  : Specifies that the subordinate automatically joins the transaction when receiving a join transaction request from the superior.

  CM_JOIN_EXPLICIT
  : Specifies that the subordinate must join the transaction explicitly when receiving a join transaction request from the superior.

- *return_code* (output)

  Specifies the result of the call execution. The *return_code* can be one of the following:

  CM_OK

  CM_CALL_NOT_SUPPORTED

  CM_PROGRAM_PARAMETER_CHECK
  : This value indicates one of the following:

    — The conversation_ID specifies an unassigned identifier.

    — The join_transaction specifies an undefined value.

    — The program is not the subordinate for the conversation.

    — The *transaction_control* characteristic is set to CM_CHAINED_TRANSACTIONS.

  CM_OPERATION_NOT_ACCEPTED
  : This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.

  CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause any state changes.

APPLICATION USAGE

1. If a *return_code* other than CM_OK is returned on the call, the *join_transaction* conversation characteristic remains unchanged.
2. This call can be issued only by the subordinate program of a conversation.
3. This call can be issued in all states except **Reset** and **Initialize**. It should be issued in the **Initialize_Incoming** state, so that it has an effect at the following Accept_Incoming call. In all the other states it is allowed only if the *transaction_control* characteristic has the value CM_UNCHAINED_TRANSACTIONS.
4. If a program wants to use CM_JOIN_EXPLICIT, it should extract the *transaction_control* characteristic after a successful Accept_Incoming call. If the value is CM_CHAINED_TRANSACTIONS, the program should join the transaction by issuing a *tx_begin*( ) call. If the value is CM_UNCHAINED_TRANSACTIONS the program is informed with a CM_JOIN_TRANSACTION *status_received* value, if it is to join the transaction. In any case, the program may first do any local work that is not to be included in the remote program's transaction before joining the transaction.
5. This call does not apply to any previous call operation still in progress.
6. This call is not associated with any conversation queue. When a conversation uses queue-level non-blocking, the call does not return CM_OPERATION_NOT_ACCEPTED on the conversation.

SEE ALSO

Section 3.14.6 on page 59 discusses how a program can join a transaction.

NAME

Set_Log_Data (CMSLD) — set the *log_data* conversation characteristic.

SYNOPSIS

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

DESCRIPTION

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 when the Initialize_Conversation, Accept_Conversation or Initialize_For_Incoming call was issued.

Note: When an LU 6.2 CRM is being used, the *log_data* characteristic is used only on basic conversations.

The Set_Log_Data (CMSLD) call uses the following input and output 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 and the conversation is using an LU 6.2 CRM.

  — The *log_data_length* specifies a value less than 0 or greater than 512.

  CM_OPERATION_NOT_ACCEPTED

  CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause a state change.

APPLICATION USAGE

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 and the conversation is using an LU 6.2 CRM

— 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 above 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.

SEE ALSO

Section 3.8.5 on page 38 provides further information on the automatic conversion of the *log_data* parameter.

Send_Error (CMSERR) on page 240 and *Deallocate (CMDEAL)* on page 147 provide further discussion on how the *log_data* characteristic is used.

NAME

Set_Mode_Name (CMSMN) — set the *mode_name* conversation characteristic.

SYNOPSIS

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

DESCRIPTION

Set_Mode_Name (CMSMN) is used by a program to set the *mode_name* and *mode_name_length* characteristics for a conversation. Set_Mode_Name overrides the current values that were 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* and *mode_name_length* characteristics 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.

The Set_Mode_Name (CMSMN) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *mode_name* (input)

  Specifies the mode name designating the network properties for the logical connection 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.

APPLICATION USAGE

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.

SEE ALSO

Section 3.5.2 on page 22 further discusses the *mode_name* conversation characteristic.

Section 3.8.5 on page 38 provides further information on the automatic conversion of the *mode_name* parameter.

Section D.3.3 on page 481 discusses SNA service transaction programs.

NAME

Set_Partner_LU_Name (CMSPLN) — set the *partner_LU_name* conversation characteristic.

SYNOPSIS

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

DESCRIPTION

Set_Partner_LU_Name (CMSPLN) is used by a program to set the *partner_LU_name* and *partner_LU_name_length* characteristics for a conversation. Set_Partner_LU_Name overrides the current values that were 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* characteristics for this conversation.

Notes:

1. 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.
2. The *partner_LU_name* characteristic is used only by an LU 6.2 CRM.

The Set_Partner_LU_Name (CMSPLN) call uses the following input and output 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_CALL_NOT_SUPPORTED

  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_PROGRAM_STATE_CHECK
  This value indicates that the conversation is not in **Initialize** state.

  CM_OPERATION_NOT_ACCEPTED

  CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause a state change.

APPLICATION USAGE

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.

SEE ALSO

Section 3.5.2 on page 22 and notes 4 and 5 of Table A-3 on page 341 provide further discussion of the *partner_LU_name* conversation characteristic.

Section 3.8.5 on page 38 provides further information on the automatic conversion of the *partner_LU_name* parameter.

NAME

Set_Prepare_Data_Permitted (CMSPDP) — set the *prepare_data_permitted* conversation characteristic.

SYNOPSIS

```
CALL CMSPDP(conversation_ID,prepare_data_permitted,return_code)
```

DESCRIPTION

Set_Prepare_Data_Permitted (CMSPDP) is used by a program to set the *prepare_data_permitted* characteristic for a given conversation. Set_Prepare_Data_Permitted overrides the value that was assigned when the Initialize_Conversation call was issued. The subordinate program on the conversation cannot issue the Set_Prepare_Data_Permitted call.

Note: The *prepare_data_permitted* characteristic is used only by an OSI TP CRM, and only for a half-duplex conversation.

The Set_Prepare_Data_Permitted (CMSPDP) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *prepare_data_permitted* (input)

  Specifies whether the superior program wants to allow the subordinate to send data following the receipt of a take-commit notification. The *prepare_data_permitted* variable can have one of the following values:

  CM_PREPARE_DATA_NOT_PERMITTED
  : Specifies the subordinate will not be permitted to send data following the receipt of a take-commit notification.

  CM_PREPARE_DATA_PERMITTED
  : Specifies the subordinate will be permitted to send data following the receipt of a take-commit notification.

- *return_code* (output)

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

  CM_OK

  CM_CALL_NOT_SUPPORTED

  CM_PROGRAM_PARAMETER_CHECK
  : This value indicates one of the following:

    — The *conversation_ID* specifies an unassigned identifier.

    — The *prepare_data_permitted* specifies CM_PREPARE_DATA_PERMITTED and the conversation is using an LU 6.2 CRM.

    — The *prepare_data_permitted* specifies an undefined value.

    — The *sync_level* is set to CM_NONE or CM_CONFIRM.

    — The program is not the superior for the conversation.

  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.

APPLICATION USAGE

1. If a *return_code* other than CM_OK is returned on the call, the *prepare_data_permitted* conversation characteristic remains unchanged.
2. When the Prepare call is issued with the *prepare_data_permitted* characteristic set to CM_PREPARE_DATA_PERMITTED, the subordinate program is notified that it is permitted to send data through a take-commit notification that ends in a *status_received* value of CM_TAKE_COMMIT_DATA_OK, CM_TAKE_COMMIT_SEND_DATA_OK or CM_TAKE_COMMIT_DEALLOC_DATA_OK.

SEE ALSO

See *Prepare (CMPREP)* on page 199 for a description of the Prepare call.

NAME

Set_Prepare_To_Receive_Type (CMSPTR) — set the *prepare_to_receive_type* conversation characteristic.

SYNOPSIS

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

DESCRIPTION

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 when the Initialize_Conversation, Accept_Conversation or Initialize_For_Incoming call was issued.

Note: The *prepare_to_receive_type* characteristic is used only for a half-duplex conversation.

The Set_Prepare_To_Receive_Type (CMSPTR) call uses the following input and output 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, or if *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM but the conversation is not currently included in a transaction, execute the function of the Flush call and enter **Receive** state.

  — If *sync_level* is set to CM_CONFIRM, or if *sync_level* is set to CM_SYNC_POINT but the conversation is not currently included in a transaction, 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 and the conversation is included in a transaction, 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.

  — If *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, and the conversation is included in a transaction, enter **Defer-Receive** state until the program issues a resource recovery commit or backout call, or until the program issues a Flush call for this conversation. If the commit 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 set to an undefined value.

  — The *prepare_to_receive_type* is CM_PREP_TO_RECEIVE_CONFIRM, but the conversation is assigned with *sync_level* set to CM_NONE or CM_SYNC_POINT_NO_CONFIRM.

  — The *send_receive_mode* of the conversation is CM_FULL_DUPLEX.

  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 a state change.

APPLICATION USAGE

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

SEE ALSO

Section 4.3.4 on page 76 shows an example program using the Prepare_To_Receive call.

Prepare_To_Receive (CMPTR) on page 202 discusses how the *prepare_to_receive_type* is used.

NAME

Set_Processing_Mode (CMSPM) — set the *processing_mode* conversation characteristic.

SYNOPSIS

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

DESCRIPTION

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 when the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call was issued. The processing mode of a conversation cannot be changed prior to the completion of all previous call operations on that conversation.

Note: The *processing_mode* characteristic is used only for a half-duplex conversation.

The Set_Processing_Mode (CMSPM) call uses the following input and output 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_CALL_NOT_SUPPORTED

  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.

    — The *send_receive_mode* of the conversation is CM_FULL_DUPLEX.

    — The program has chosen queue-level non-blocking for the conversation.

CM_OPERATION_NOT_ACCEPTED
This value indicates that a previous call operation on the conversation is still in progress.

CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause any state changes.

APPLICATION USAGE

1. A program can choose to use conversation-level non-blocking by issuing the Set_Processing_Mode call to set the *processing_mode* characteristic to CM_NON_BLOCKING for a conversation. The processing mode applies to all the subsequent calls on that conversation until it is set otherwise or the conversation ends.
2. If a *return_code* other than CM_OK is returned on the call, the *processing_mode* conversation characteristic is unchanged.
3. When CM_OPERATION_INCOMPLETE is returned from any of the calls listed in Table 3-6 on page 43, 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. For conversations using conversation-level non-blocking, the Wait_For_Conversation call is used to determine when an operation is completed. Each call to Wait_For_Conversation returns 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.

   If programs place the Wait_For_Conversation call in a procedure other than the accompanying CPI-C call that returned CM_OPERATION_INCOMPLETE, all parameters of that CPI-C call must be in a global storage and not in automatic storage. This is important even for parameters that the program won't use. For example if the partner program never uses any call that results in a *control_information_received* value other than CM_NO_CONTROL_INFO_RECEIVED, the local program nevertheless has to place the *control_information_received* parameter of the Receive call in the global storage.
4. Not all language processors support the use of non-blocking operations. See Section 5.2 on page 117 for language processor restrictions on the use of non-blocking operations.

SEE ALSO

Section 3.9 on page 40 discusses the use of concurrent operations and conversation queues.

Section 3.10 on page 43 discusses the use of non-blocking operations.

Section 4.3.8 on page 84 shows an example of a program that uses conversation-level non-blocking calls to accept multiple incoming half-duplex conversations.

Set_Queue_Callback_Function (CMSQCF) on page 297 describes how to set a callback function and related information for a conversation queue.

Set_Queue_Processing_Mode (CMSQPM) on page 300 describes how to set the processing mode for a conversation queue.

Wait_For_Conversation (CMWAIT) on page 325 describes the use of Wait_For_Conversation to wait for completion of a conversation-level outstanding operation.

NAME

Set_Queue_Callback_Function (CMSQCF) — set a callback function and a user field for a given conversation queue, and set the queue's processing mode to CM_NON_BLOCKING.

SYNOPSIS

```
CALL CMSQCF(conversation_ID,conversation_queue,callback_function,
                user_field,return_code)
```

DESCRIPTION

Set_Queue_Callback_Function (CMSQCF) is used to set a callback function and a user field for a given conversation queue, and to set the queue's processing mode to CM_NON_BLOCKING.

The Set_Queue_Callback_Function (CMSQCF) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *conversation_queue* (input)

  Specifies the conversation queue on which completion of a call operation will invoke the callback function. The *conversation_queue* can have one of the following values:

  CM_INITIALIZATION_QUEUE

  CM_SEND_QUEUE

  CM_RECEIVE_QUEUE

  CM_SEND_RECEIVE_QUEUE

  CM_EXPEDITED_SEND_QUEUE

  CM_EXPEDITED_RECEIVE_QUEUE.

- *callback_function* (input)

  Specifies a callback function to be set for the identified queue.

- *user_field* (input)

  Specifies a user field to be associated with the identified queue.

- *return_code* (output)

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

  CM_OK

  CM_CALL_NOT_SUPPORTED

  CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:

  — The *conversation_ID* contains an unassigned conversation identifier.

  — The *conversation_queue* specifies a value that is not defined for the *send_receive_mode* conversation characteristic.

  — The program has chosen conversation-level non-blocking for the conversation.

CM_PROGRAM_STATE_CHECK
This value indicates one of the following:

— The *conversation_queue* parameter is set to CM_INITIALIZATION_QUEUE, and the conversation is not in **Initialize** or **Initialize-Incoming** state.

— The *conversation_queue* parameter is set to CM_SEND_QUEUE, CM_RECEIVE_QUEUE, CM_SEND_RECEIVE_QUEUE, CM_EXPEDITED_SEND_QUEUE or CM_EXPEDITED_RECEIVE_QUEUE, and the conversation is in **Initialize-Incoming** state.

CM_OPERATION_NOT_ACCEPTED

CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause a state change.

APPLICATION USAGE

1. Because of requiring support of passing a callback function as a parameter, the call is supported by the C programming language only.
2. A program can choose to use queue-level non-blocking by issuing the Set_Queue_Callback_Function call (or the Set_Queue_Processing_Mode call) for a conversation queue. When the call completes successfully, the *processing_mode* characteristic becomes meaningless to the conversation.
3. The call is associated with the queue specified in the *conversation_queue* parameter.
4. The program can issue the call for a conversation queue that is defined for the current send-receive mode. The defined queues for each send-receive mode are listed in Table 5-2 on page 302 under the **APPLICATION USAGE** section of the Set_Queue_Processing_Mode call.

   In the special case when the conversation is in **Initialize-Incoming** state, the *send_receive_mode* characteristic has no defined value. The program can issue the call only for the Initialization queue.
5. Until the program sets the processing mode for a conversation queue (or chooses conversation-level non-blocking), all the calls associated with that queue are processed in blocking mode.
6. The call sets the processing mode of the identified queue to CM_NON_BLOCKING. The processing mode applies to all subsequent calls to the queue until the processing mode is set to CM_BLOCKING using a Set_Queue_Processing_Mode (CMSQPM) call or the conversation ends.
7. Once set for the identified queue, the callback function and user field will be associated with all subsequent outstanding operations on the queue until they are set differently, a Set_Queue_Processing_Mode (CMSQPM) call is issued, or the conversation ends.
8. When CM_OPERATION_INCOMPLETE is returned from any of the calls listed in Table 3-6 on page 43, 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.
9. A callback function is a user-defined routine and has two input parameters: *user_field* and *call_ID*. It is used to handle completion of an outstanding operation.

If a callback function is set for a conversation queue, the function is invoked when an outstanding operation on the queue completes. The user field as specified by the program, and call ID for the completed operation, can then be passed to the callback function.

Note: An exception is the Microsoft Windows environment. The *callback_function* parameter is ignored in Windows, and the *user_field* parameter contains a pointer pointing to a structure of type **MSG**, as defined by Windows. The structure contains at least a window handle, message number, word value, and doubleword value. When an outstanding operation completes, CPI Communications uses these fields as arguments to call the Windows **PostMessage** function. Upon catching the message, the program can then invoke a routine (taking the word value and doubleword value as input) to handle the completion of the outstanding operation.

SEE ALSO

Section 3.9 on page 40 discusses the use of concurrent operations and conversation queues.

Section 3.10 on page 43 discusses the use of non-blocking operations.

Set_Queue_Processing_Mode (CMSQPM) on page 300 describes how to set the processing mode for a conversation queue.

Wait_For_Completion (CMWCMP) on page 322 describes the use of Wait_For_Completion to wait for completion of an outstanding operation on a conversation queue.

NAME

Set_Queue_Processing_Mode (CMSQPM) — set the processing mode for a given conversation queue and to associate an outstanding-operation identifier (OOID) and a user field with the queue.

SYNOPSIS

```
CALL CMSQPM(conversation_ID,conversation_queue,queue_processing_mode,
                    user_field,OOID,return_code)
```

DESCRIPTION

Set_Queue_Processing_Mode (CMSQPM) is used to set the processing mode for a given conversation queue. When the *queue_processing_mode* is set to CM_NON_BLOCKING, this call also associates an outstanding-operation identifier (OOID) and a user field with the queue.

The Set_Queue_Processing_Mode (CMSQPM) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *conversation_queue* (input)

  Specifies the conversation queue for which the processing mode is to be set by this call. *conversation_queue* can have one of the following values:

  CM_INITIALIZATION_QUEUE

  CM_SEND_QUEUE

  CM_RECEIVE_QUEUE

  CM_SEND_RECEIVE_QUEUE

  CM_EXPEDITED_SEND_QUEUE

  CM_EXPEDITED_RECEIVE_QUEUE.

- *queue_processing_mode* (input)

  Specifies the processing mode to be used for the identified queue. *queue_processing_mode* can have one of the following values:

  CM_BLOCKING

  CM_NON_BLOCKING.

- *user_field* (input)

  Specifies a user field to be associated with the identified queue, when *queue_processing_mode* is set to CM_NON_BLOCKING.

- *OOID* (output)

  Specifies an outstanding operation identifier assigned to the identified queue, when *queue_processing_mode* is set to CM_NON_BLOCKING.

- *return_code* (output)

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

  CM_OK

CM_CALL_NOT_SUPPORTED

CM_PROGRAM_PARAMETER_CHECK
This value indicates one of the following:

— The *conversation_ID* contains an unassigned conversation identifier.

— The *conversation_queue* specifies a value that is not defined for the *send_receive_mode* conversation characteristic.

— The *queue_processing_mode* specifies an undefined value.

— The program has chosen conversation-level non-blocking for the conversation.

CM_PROGRAM_STATE_CHECK
This value indicates one of the following:

— The *conversation_queue* parameter is set to CM_INITIALIZATION_QUEUE, and the conversation is not in **Initialize** or **Initialize-Incoming** state.

— The *conversation_queue* parameter is set to CM_SEND_QUEUE, CM_RECEIVE_QUEUE, CM_SEND_RECEIVE_QUEUE, CM_EXPEDITED_SEND_QUEUE or CM_EXPEDITED_RECEIVE_QUEUE, and the conversation is in **Initialize-Incoming** state.

CM_OPERATION_NOT_ACCEPTED

CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause a state change.

APPLICATION USAGE

1. A program can choose to use queue-level non-blocking by issuing the Set_Queue_Processing_Mode call (or the Set_Queue_Callback_Function call) for a conversation queue. When the call completes successfully, the *processing_mode* characteristic becomes meaningless to the conversation.
2. The call is associated with the queue specified in the *conversation_queue* parameter.
3. The program can issue the call for a conversation queue that is defined for the current send-receive mode. The defined queues for each send-receive mode are listed in Table 5-2 on page 302.

   In the special case when the conversation is in **Initialize-Incoming** state, the *send_receive_mode* characteristic has no defined value. The program can issue the call only for the Initialization queue.

Table 5-2 Full-duplex and Half-duplex Conversation Queues

Send-Receive Mode	Conversation Queues
Full-duplex	Initialization Send Receive Expedited-Send Expedited-Receive
Half-duplex	Initialization Send-Receive Expedited-Send Expedited-Receive

4. Until the program sets the processing mode for a conversation queue (or chooses conversation-level non-blocking), all the calls associated with that queue are processed in blocking mode.
5. If a return code other than CM_OK is returned on the call, the processing mode of the specified queue is unchanged.
6. Once a processing mode is set for a conversation queue, the processing mode applies to all the subsequent calls associated with the queue until it is set differently using the call or until the conversation ends.
7. If the queue processing mode is CM_BLOCKING, no OOID is returned on the call. Therefore, the program should ignore the *OOID* parameter.
8. If the queue processing mode is CM_NON_BLOCKING, an OOID is returned on the call. The OOID will be associated with all subsequent outstanding operations on the queue until the processing mode is set to CM_BLOCKING, a Set_Queue_Callback_Function (CMSQCF) call is issued, or the conversation ends.
9. When the program issues the call for a conversation queue and sets the queue processing mode to CM_NON_BLOCKING for the first time, an OOID is created and set for the queue. The OOID remains with the queue until the conversation ends. Even if the queue processing mode changes several times or a Set_Queue_Callback_Function (CMSQCF) call is issued during the conversation, each Set_Queue_Processing_Mode call to return to CM_NON_BLOCKING reactivates the same OOID for that queue.
10. When CM_OPERATION_INCOMPLETE is returned from any of the calls listed in Table 3-6 on page 43, 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. For conversations using the Set_Queue_Processing_Mode call, the Wait_For_Completion call is used to determine when an outstanding operation is completed.
11. The program may specify a user field on the call when the queue processing mode is CM_NON_BLOCKING. If the program chooses to do so, the user field will be associated with the identified queue along with an OOID. The user field specified by the program will be returned on the *user_field_list* parameter of the Wait_For_Completion (CMWCMP) call when an outstanding operation with the OOID has completed.

SEE ALSO

Section 3.9 on page 40 discusses the use of concurrent operations and conversation queues.

Section 3.10 on page 43 discusses the use of non-blocking operations.

Section 4.3.12 on page 92 shows an example of a program that uses queue-level non-blocking.

Set_Queue_Callback_Function (CMSQCF) on page 297 describes how to set a callback function and related information for a conversation queue.

Wait_For_Completion (CMWCMP) on page 322 describes the use of Wait_For_Completion to wait for completion of an outstanding operation on a conversation queue.

NAME

Set_Receive_Type (CMSRT) — set the *receive_type* conversation characteristic.

SYNOPSIS

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

DESCRIPTION

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 when the Initialize_Conversation, Accept_Conversation or Initialize_For_Incoming call was issued.

The Set_Receive_Type (CMSRT) call uses the following input and output 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.

APPLICATION USAGE

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

SEE ALSO

Section 4.3.2 on page 72 discusses how a program can use Set_Receive_Type with a value of CM_RECEIVE_IMMEDIATE.

Receive (CMRCV) on page 208 discusses how the *receive_type* characteristic is used.

NAME

Set_Return_Control (CMSRC) — set the *return_control* conversation characteristic.

SYNOPSIS

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

DESCRIPTION

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 when the Initialize_Conversation call was issued.

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.

The Set_Return_Control (CMSRC) call uses the following input and output 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 logical connection for the conversation before returning control to the program.

  CM_IMMEDIATE
  : Allocate a logical connection for the conversation if a logical connection is immediately available and return control to the program with one of the following return codes indicating whether or not a logical connection is allocated.

    — A return code of CM_OK indicates a logical connection was immediately available and has been allocated for the conversation. A logical connection is immediately available when it is active; the logical connection is not allocated to another conversation; and, for an LU 6.2 CRM, the local system is the contention winner for the logical connection.

    — A return code of CM_UNSUCCESSFUL indicates a logical connection 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.

APPLICATION USAGE

1. An allocation error resulting from the local system's failure to obtain a logical connection for the conversation is reported on the Allocate call. An allocation error resulting from the remote system's rejection of the conversation startup request is reported on a subsequent conversation call.
2. For an LU 6.2 CRM, two systems connected by a logical connection may both attempt to allocate a conversation on the logical connection 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 the referenced SNA Programmer's Reference specification.

   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.

SEE ALSO

Allocate (CMALLC) on page 130 provides more discussion on the use of the *return_control* characteristic in allocating a conversation.

NAME

Set_Send_Receive_Mode (CMSSRM) — set the *send_receive_mode* conversation characteristic.

SYNOPSIS

```
CALL CMSSRM(conversation_ID,send_receive_mode,return_code)
```

DESCRIPTION

The Set_Send_Receive_Mode (CMSSRM) call is used by a program to set the *send_receive_mode* characteristic for a conversation. Set_Send_Receive_Mode overrides the value that was assigned when the Initialize_Conversation call was issued.

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

The Set_Send_Receive_Mode (CMSSRM) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *send_receive_mode* (input)

  Specifies the send-receive mode of the conversation.

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

  CM_HALF_DUPLEX
  : Specifies the allocation of a half-duplex conversation.

  CM_FULL_DUPLEX
  : Specifies the allocation of a full-duplex 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_CALL_NOT_SUPPORTED

  CM_PROGRAM_STATE_CHECK
  : This value indicates that the program 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 *send_receive_mode* specifies an undefined value.

    — The *sync_level* is set to CM_CONFIRM or CM_SYNC_POINT, and *send_receive_mode* is set to CM_FULL_DUPLEX.

    — The *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, the conversation is using a LU 6.2 CRM, and the *send_receive_mode* specifies CM_HALF_DUPLEX.

    — The *send_type* is set to CM_SEND_AND_CONFIRM or CM_SEND_AND_PREP_TO_RECEIVE, and *send_receive_mode* is set to CM_FULL_DUPLEX.

    — The *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, the conversation is using a LU 6.2 CRM, and the *send_receive_mode* specifies CM_FULL_DUPLEX.

— The program has selected conversation-level non-blocking by issuing Set_Processing_Mode successfully, and *send_receive_mode* is set to CM_FULL_DUPLEX.

CM_OPERATION_NOT_ACCEPTED

CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause a state change.

APPLICATION USAGE

1. If a *return_code* other than CM_OK is returned on the call, the *send_receive_mode* characteristic is unchanged.
2. Set_Send_Receive_Mode overrides the value assigned with the Initialize_Conversation call and can only be issued when the program is in **Initialize** state.

SEE ALSO

Section 3.3 on page 19 provides more information on the differences between half-duplex and full-duplex conversations.

Section 4.3.9 on page 86 shows an example of how a full-duplex conversation is set up.

Extract_Send_Receive_Mode (CMESRM) on page 182 tells how to determine the send-receive mode used for a conversation.

NAME

Set_Send_Type (CMSST) — set the *send_type* conversation characteristic.

SYNOPSIS

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

DESCRIPTION

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 when the Initialize_Conversation, Accept_Conversation or Initialize_For_Incoming call was issued.

The Set_Send_Type (CMSST) call uses the following input and output 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 (half-duplex conversations only)
  : 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 (half-duplex conversations only)
  : 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.

    Note: The action depends on the value of the *prepare_to_receive_type* characteristic for the conversation.

  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.

    Note: The action depends on the value of the *deallocate_type* characteristic for 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_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 or CM_SYNC_POINT_NO_CONFIRM.

  — The *send_type* specifies an undefined value.

  — The *send_type* is set to CM_SEND_AND_CONFIRM or CM_SEND_AND_PREP_TO_RECEIVE and the *send_receive_mode* is set to CM_FULL_DUPLEX.

  CM_PROGRAM_STATE_CHECK
  This value indicates the conversation is in **Initialize-Incoming** state.

  CM_OPERATION_NOT_ACCEPTED

  CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause a state change.

APPLICATION USAGE

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

SEE ALSO

Section 4.3.4 on page 76 shows an example program flow using the Set_Send_Type call.

Send_Data (CMSEND) on page 230 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:

- *Confirm (CMCFM)* on page 137
- *Deallocate (CMDEAL)* on page 147
- *Flush (CMFLUS)* on page 190
- *Prepare_To_Receive (CMPTR)* on page 202.

NAME

Set_Sync_Level (CMSSL) — set the *sync_level* conversation characteristic.

SYNOPSIS

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

DESCRIPTION

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 when the Initialize_Conversation call was issued.

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.

The Set_Sync_Level (CMSSL) call uses the following input and output 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 (half-duplex conversations only)
  : The programs can perform confirmation processing on this conversation. The programs can issue calls and recognize returned parameters relating to confirmation.

  CM_SYNC_POINT (half-duplex conversations only)
  : 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.

  CM_SYNC_POINT_NO_CONFIRM
  : 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 cannot perform confirmation processing.

  Note: If the conversation is using an OSI TP CRM, confirmation of the deallocation of the conversation can be performed with any *sync_level* value.

- *return_code* (output)

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

  CM_OK

  CM_PARM_VALUE_NOT_SUPPORTED
  This value indicates that the *sync_level* specifies CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and the value is not supported by the local system.

  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* specifies CM_NONE, the *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, and the conversation is using an LU 6.2 CRM.

  — The *sync_level* specifies CM_NONE, the *send_receive_mode* is set to CM_HALF_DUPLEX, and the *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_CONFIRM.

  — The *sync_level* specifies CM_NONE or CM_SYNC_POINT_NO_CONFIRM, the *send_receive_mode* is set to CM_HALF_DUPLEX, and the *send_type* is set to CM_SEND_AND_CONFIRM.

  — The *sync_level* specifies CM_CONFIRM or CM_SYNC_POINT and the *send_receive_mode* is set to CM_FULL_DUPLEX.

  — The *sync_level* specifies CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the *deallocate_type* is set to CM_DEALLOCATE_FLUSH or CM_DEALLOCATE_CONFIRM, and the conversation is using an LU 6.2 CRM.

  — The *sync_level* specifies CM_SYNC_POINT_NO_CONFIRM, the *send_receive_mode* is set to CM_HALF_DUPLEX, and the conversation is using an LU 6.2 CRM.

  — The *sync_level* specifies an undefined value.

  CM_OPERATION_NOT_ACCEPTED

  CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause a state change.

APPLICATION USAGE

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

SEE ALSO

Section 4.3.3 on page 74 shows how to use the Set_Sync_Level call.

Confirm (CMCFM) on page 137 and *Confirmed (CMCFMD)* on page 141 provide further information on confirmation processing.

NAME

Set_TP_Name (CMSTPN) — set the *TP_name* conversation characteristic.

SYNOPSIS

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

DESCRIPTION

Set_TP_Name (CMSTPN) is used by a program to set the *TP_name* and *TP_name_length* characteristics for a given conversation. Set_TP_Name overrides the current values that were originally acquired from the side information using the *sym_dest_name*. See Section 3.5.2 on page 22 for more information. This call does not change the values in the side information. Set_TP_Name only changes the *TP_name* and *TP_name_length* characteristics 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.

The Set_TP_Name (CMSTPN) call uses the following input and output 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 Section D.3.3 on page 481.)

- *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.

APPLICATION USAGE

1. Specify *TP_name* using the local system's native encoding. CPI Communications automatically converts the *TP_name* from the native 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.

SEE ALSO

Section 3.5.2 on page 22 and the notes following Table A-3 on page 341 provide further discussion of the *TP_name* conversation characteristic.

Section 3.8.5 on page 38 provides further information on the automatic conversion of the *TP_name* parameter.

Section D.3.3 on page 481 provides more information on privilege and service transaction programs.

NAME

Set_Transaction_Control (CMSTC) — set the *transaction_control* conversation characteristic.

SYNOPSIS

```
CALL CMSTC(conversation_ID,transaction_control,return_code)
```

DESCRIPTION

Set_Transaction_Control (CMSTC) is used by a program to set the *transaction_control* characteristic for a given conversation. Set_Transaction_Control overrides the value that was assigned when the Initialize_Conversation call was issued.

Notes:

1. Only the program that initiates the conversation can issue this call.
2. The *transaction_control* characteristic is used only by an OSI TP CRM.

The Set_Transaction_Control (CMSTC) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *transaction_control* (input)

  Specifies whether the superior program wants to use chained or unchained transactions on the conversation with the subordinate. The *transaction_control* variable can have one of the following values:

  CM_CHAINED_TRANSACTIONS
  Specifies that the conversation will use chained transactions.

  CM_UNCHAINED_TRANSACTIONS
  Specifies that the conversation will use unchained transactions.

- *return_code* (output)

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

  CM_OK

  CM_CALL_NOT_SUPPORTED

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

  CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:

  — The *conversation_ID* specifies an unassigned identifier.

  — The *transaction_control* specifies an undefined value.

  — The *sync_level* is set to either CM_NONE or CM_CONFIRM.

  — The *transaction_control* specifies CM_UNCHAINED_TRANSACTIONS, and the conversation is using an LU 6.2 CRM.

  CM_OPERATION_NOT_ACCEPTED

  CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause any state changes.

APPLICATION USAGE

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

SEE ALSO

Section 3.14.5 on page 58 provides more information about using chained and unchained transactions with CPI Communications.

NAME

Specify_Local_TP_Name (CMSLTP) — associate a name with itself.

SYNOPSIS

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

DESCRIPTION

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 simultaneously. It can extract the *TP_name* for a particular conversation using the Extract_TP_Name call.

The Specify_Local_TP_Name (CMSLTP) call uses the following input and output 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_CALL_NOT_SUPPORTED

  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.

APPLICATION USAGE

1. If a *return_code* other than CM_OK is returned on the call, the names associated with the current program remain unchanged.
2. Any of the names associated with the program at the time an Accept_Conversation or Accept_Incoming call is issued can be used to satisfy the call.
3. 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 is added to the names used to satisfy subsequent Accept_Incoming or Accept_Conversation calls.

SEE ALSO

Accept_Conversation (CMACCP) on page 125 and *Accept_Incoming (CMACCI)* on page 127 describe how an incoming conversation is accepted by a program.

Release_Local_TP_Name (CMRLTP) on page 226 explains additional timing considerations for names associated with the program.

NAME

Test_Request_To_Send_Received (CMTRTS) — determine whether or not the remote program is requesting to send data.

SYNOPSIS

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

DESCRIPTION

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

Note: The Test_Request_To_Send_Received call has meaning only when a half-duplex conversation is being used.

The Test_Request_To_Send_Received (CMTRTS) call uses the following input and output parameters:

- *conversation_ID* (input)

  Specifies the conversation identifier.

- *control_information_received* (output)

  Specifies the variable containing an indication of whether or not control information has been received.

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

  CM_NO_CONTROL_INFO_RECEIVED
  : Indicates that no control information was received.

  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 227 for further discussion of the local program's possible responses.

  CM_ALLOCATE_CONFIRMED (OSI TP CRM only)
  : The local program received confirmation of the remote program's acceptance of the conversation.

  CM_ALLOCATE_CONFIRMED_WITH_DATA (OSI TP CRM only)
  : The local program received confirmation of the remote program's acceptance of the conversation. The local program may now issue an Extract_Initialization_Data (CMEID) call to receive the initialization data.

  CM_ALLOCATE_REJECTED_WITH_DATA (OSI TP CRM only)
  : The remote program rejected the conversation. The local program may now issue an Extract_Initialization_Data (CMEID) call to receive the initialization data.

    This value is returned with a return code of CM_OK. The program will receive a CM_DEALLOCATED_ABEND return code on a later call on the conversation.

  CM_EXPEDITED_DATA_AVAILABLE (LU 6.2 CRM only)
  : Expedited data is available to be received.

  CM_RTS_RCVD_AND_EXP_DATA_AVAIL (LU 6.2 CRM only)
  : The local program received a request-to-send notification from the remote program and expedited data is available to be received.

Notes:

1. If *return_code* is set to CM_PROGRAM_PARAMETER_CHECK or CM_PROGRAM_STATE_CHECK, the value contained in *control_information_received* has no meaning.
2. When more than one piece of control information is available to be returned to the program, it will be returned in the following order:
   - CM_ALLOCATE_CONFIRMED, CM_ALLOCATE_CONFIRMED_WITH_DATA or CM_ALLOCATE_REJECTED_WITH_DATA
   - CM_RTS_RCVD_AND_EXP_DATA_AVAIL
   - CM_REQ_TO_SEND_RECEIVED
   - CM_EXPEDITED_DATA_AVAILABLE
   - CM_NO_CONTROL_INFO_RECEIVED.

- *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 or CM_SYNC_POINT_NO_CONFIRM, the program is in the **Backout-Required** condition. The Test_Request_To_Send_Received call is not allowed for this conversation while the program is in this condition.

  CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:

  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *send_receive_mode* of the conversation is CM_FULL_DUPLEX.

  CM_OPERATION_NOT_ACCEPTED
  This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.

  CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause a state change.

APPLICATION USAGE

1. When the local system receives a request-to-send or allocate-confirmed notification, it retains the notification until the local program issues a call (such as Test_Request_To_Send_Received) with the *control_information_received* parameter. It retains one request-to-send or allocate-confirmed 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.

When the local system receives expedited data from the partner system, it is indicated on the calls that have the *control_information_received* parameter until the expedited data is actually received by the program.

2. After the retained notification, other than the expedited data notification, is indicated to the local program through the *control_information_received* parameter, the local system discards the notification.

3. Implementors should note that a request-to-send or allocate-confirmed notification can be reported on this call (not associated with any queue), on the Send_Expedited_Data call (associated with the Expedited-Send queue), and on the Receive_Expedited_Data call (associated with the Expedited-Receive queue). When the program uses queue-level non-blocking, more than one of these calls may be executed simultaneously. An implementation should report the notification to the program only once, through one of these calls.

4. A program should not rely solely on this call to test whether expedited data is available. Expedited data may be available in the CRM but the implementation of the CPIC layer may not always be able to indicate this to the program on this call. To test for the availability of expedited data, the program should issue Receive_Expedited_Data with the *expedited_receive_type* set to CM_RECEIVE_IMMEDIATE.

SEE ALSO

Request_To_Send (CMRTS) on page 227 provides further discussion of the request-to-send function, and *Set_Allocate_Confirm (CMSAC)* on page 255 provides more information about the allocate-confirmed function.

NAME

Wait_For_Completion (CMWCMP) — wait for completion of one or more outstanding operations represented in a specified outstanding-operation-ID (OOID) list.

SYNOPSIS

```
CALL CMWCMP(OOID_list,OOID_list_count,timeout,completed_op_index_list,
            completed_op_count,user_field_list,return_code)
```

DESCRIPTION

Wait_For_Completion (CMWCMP) is used to wait for completion of one or more outstanding operations represented in a specified outstanding-operation-ID (OOID) list or its count.

The Wait_For_Completion (CMWCMP) call uses the following input and output parameters:

- *OOID_list* (input)

  Specifies a list of OOIDs representing the outstanding operations for which completion is expected.

- *OOID_list_count* (input)

  Specifies the number of OOIDs contained in *OOID_list*.

- *timeout* (input)

  Specifies the amount of time in milliseconds that the program is willing to wait for completion of an operation. Valid *timeout* values are zero or any greater integer number.

- *completed_op_index_list* (output)

  Specifies a list of indexes corresponding to the OOIDs in *OOID_list* for which the associated operations have completed. The index is the position of an OOID in *OOID_list*, beginning with 1.

- *completed_op_count* (output)

  Specifies the number of indexes contained in *completed_op_index_list*, or the number of user fields contained in *user_field_list*, or both.

- *user_field_list* (output)

  Specifies a list of user fields corresponding to the completed operations.

- *return_code* (output)

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

  CM_OK

  CM_CALL_NOT_SUPPORTED

  CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:

  — The *OOID_list_count* specifies a value less than 1.

  — The number of OOIDs in *OOID_list* is less than the value specified in *OOID_list_count*.

  — The *OOID_list* contains an unassigned OOID.

  — The *timeout* specifies a value less than zero.

CM_PROGRAM_STATE_CHECK
: This value indicates that there is no outstanding operation associated with any of the OOIDs specified in *OOID_list* or by use of a defined value of *OOID_list_count*.

CM_UNSUCCESSFUL
: This value indicates that the specified timeout value has elapsed and none of the operations specified in *OOID_list* or by use of a defined value of *OOID_list_count* has completed.

CM_PRODUCT_SPECIFIC_ERROR.

STATE CHANGES

This call does not cause a state change.

APPLICATION USAGE

1. Unless the *return_code* indicates CM_OK, the values of all other parameters on this call have no meaning.
2. The call returns the OOID corresponding to a completed operation only once. At that time, all information about the completed operation is purged from the associated queue. If the program issues the call to check the status of the same operation again, the OOID is not returned.
3. When the call returns a completion operation to the program, the return code of the completion operation can be found in the *return_code* parameter on the completed call.
4. A special timeout value of zero can be used to check the status of all the operations whose OOIDs are specified in the *OOID_list* parameter. The call specified in this way incurs no blocking.
5. The program can replace a previously returned OOID in the *OOID_list* parameter with a null OOID (integer zero) and continue to use the same list for the following Wait_For_Completion call. The null OOID is not associated with any outstanding operation.
6. There is a one-to-one correspondence between elements of the completed-operation-index list and those of the user-field list. Hence, the size of the user-field list equals that of the completed-operation-index list.
7. The program should allocate the same amount of storage for the completed-operation-index list and user-field list as it does for the outstanding-operation-ID list. If there is not enough storage allocated, the program may lose some OOIDs and user fields that correspond to the completed operations.
8. In a multi-threaded environment, concurrent Wait_For_Completion operations can occur. If an OOID is specified on more than one Wait_For_Completion call, the OOID is returned on only one of the Wait_For_Completion calls when the corresponding outstanding operation completes.
9. Implementors should note that after returning CM_OPERATION_INCOMPLETE to the program, an implementation should not fill the return code for the outstanding operation before the program checks the return-code value. It is recommended that implementations fill the return code only when the program issues a Wait_For_Completion call for the outstanding operation.

SEE ALSO

Section 3.10 on page 43 discusses the use of non-blocking operations.

Section 4.3.8 on page 84 shows an example of a program that uses conversation-level non-blocking calls to accept multiple incoming half-duplex conversations.

Set_Queue_Processing_Mode (CMSQPM) on page 300 describes how to set the processing mode for a conversation queue.

NAME

Wait_For_Conversation (CMWAIT) — wait for the completion of any conversation-level outstanding operation.

SYNOPSIS

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

DESCRIPTION

A program must use the Wait_For_Conversation (CMWAIT) call to wait for the completion of an outstanding operation on a conversation using conversation-level non-blocking. 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, Receive_Expedited_Data, Request_To_Send, Send_Data, Send_Error, or Send_Expedited_Data call. This can occur when the *processing_mode* conversation characteristic is set to CM_NON_BLOCKING and the requested operation cannot complete immediately.

The Wait_For_Conversation (CMWAIT) call uses the following output 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_BUFFER_TOO_SMALL

  CM_CALL_NOT_SUPPORTED

  CM_CONV_DEALLOC_AFTER_SYNCPT

  CM_CONVERSATION_ENDING

  CM_CONVERSATION_TYPE_MISMATCH

  CM_DEALLOC_CONFIRM_REJECT

  CM_DEALLOCATED_ABEND

  CM_DEALLOCATED_ABEND_BO

  CM_DEALLOCATED_ABEND_SVC

  CM_DEALLOCATED_ABEND_SVC_BO

  CM_DEALLOCATED_ABEND_TIMER

  CM_DEALLOCATED_ABEND_TIMER_BO

  CM_DEALLOCATED_NORMAL

CM_DEALLOCATED_NORMAL_BO

CM_EXP_DATA_NOT_SUPPORTED

CM_INCLUDE_PARTNER_REJECT_BO

CM_PIP_NOT_SPECIFIED_CORRECTLY

CM_PRODUCT_SPECIFIC_ERROR

CM_PROGRAM_ERROR_NO_TRUNC

CM_PROGRAM_ERROR_PURGING

CM_PROGRAM_ERROR_TRUNC

CM_RESOURCE_FAIL_NO_RETRY_BO

CM_RESOURCE_FAILURE_NO_RETRY

CM_RESOURCE_FAILURE_RETRY

CM_RESOURCE_FAILURE_RETRY_BO

CM_RETRY_LIMIT_EXCEEDED

CM_SECURITY_NOT_SUPPORTED

CM_SECURITY_NOT_VALID

CM_SEND_RCV_MODE_NOT_SUPPORTED

CM_SVC_ERROR_NO_TRUNC

CM_SVC_ERROR_PURGING

CM_SVC_ERROR_TRUNC

CM_SYNC_LVL_NOT_SUPPORTED_PGM

CM_SYNC_LVL_NOT_SUPPORTED_SYS

CM_TAKE_BACKOUT

CM_TP_NOT_AVAILABLE_NO_RETRY

CM_TP_NOT_AVAILABLE_RETRY

CM_TPN_NOT_RECOGNIZED.

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 conversation-level 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.

APPLICATION USAGE

1. Wait_For_Conversation waits for the completion of any outstanding operation on any conversation using conversation-level non-blocking. 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. In a multi-threaded environment, concurrent operations may occur. A Wait_For_Conversation call waits for any operation, on any conversation using conversation-level non-blocking, that either is already outstanding or becomes outstanding during execution of the Wait_For_Conversation call. In case of concurrent Wait_For_Conversation operations, completion of an outstanding operation is indicated on one Wait_For_Conversation call only.

   Note: An implementation should serialize execution of the concurrent Wait_For_Conversation operations to prevent any Wait_For_Conversation call from hanging forever.
3. 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.
4. This call applies only to conversations using conversation-level non-blocking support.

SEE ALSO

Section 3.10 on page 43 discusses the use of non-blocking operations.

Section 4.3.8 on page 84 shows an example of a program that uses conversation-level non-blocking calls to accept multiple incoming half-duplex conversations.

Cancel_Conversation (CMCANC) on page 135 describes the means for terminating an operation before it is completed.

Set_Processing_Mode (CMSPM) on page 295 describes setting the *processing_mode* conversation characteristic.

Appendix A

Variables and Characteristics

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

- A table showing the values that variables and characteristics can take. The valid pseudonyms and corresponding integer values are provided for each variable and characteristic. (See Section A.1 on page 330.)
- The character sets used by CPI Communications. (See Section A.2 on page 337.)
- The data definitions for types and lengths of all CPI Communications characteristics and variables. (See Section A.3 on page 340.)

A.1 Pseudonyms and Integer Values

As explained in Section 1.3 on page 3, the values for variables and conversation characteristics are 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 specification shows the *return_code* being set to a pseudonym value of CM_OK. Table A-1 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. See Appendix F 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 provides a more detailed description of its values, in addition to the list of values provided here.

Table A-1 Variables / Characteristics and their Possible Values

Variable or CharacteristicName	Pseudonym Values	Integer Values
AE_qualifier_format	CM_DN	0
	CM_INT_DIGITS	2
allocate_confirm	CM_ALLOCATE_NO_CONFIRM	0
	CM_ALLOCATE_CONFIRM	1
AP_title_format	CM_DN	0
	CM_OID	1
begin_transaction	CM_BEGIN_IMPLICIT	0
	CM_BEGIN_EXPLICIT	1
call_ID †	CM_CMACCI	1
	CM_CMACCP	2
	CM_CMALLC	3
	CM_CMCANC	4
	CM_CMCFM	5
	CM_CMCFMD	6
	CM_CMCNVI	7
	CM_CMCNVO	8
	CM_CMDEAL	9
	CM_CMDFDE	10
	CM_CMEACN	11
	CM_CMEAEQ	12
	CM_CMEAPT	13
	CM_CMECS	14
	CM_CMECT	15
	CM_CMEID	17

Variable or CharacteristicName	Pseudonym Values	Integer Values
	CM_CMEMBS	18
	CM_CMEMN	19
	CM_CMEPLN	21
	CM_CMESI	22
	CM_CMESL	23
	CM_CMESRM	24
	CM_CMESUI	25
	CM_CMETC	26
	CM_CMETPN	27
	CM_CMFLUS	28
	CM_CMINCL	29
	CM_CMINIC	30
	CM_CMINIT	31
	CM_CMPREP	32
	CM_CMPTR	33
	CM_CMRCV	34
	CM_CMRCVX	35
	CM_CMRLTP	36
	CM_CMRTS	37
	CM_CMSAC	38
	CM_CMSACN	39
	CM_CMSAEQ	40
	CM_CMSAPT	41
	CM_CMSBT	42
	CM_CMSCSP	43
	CM_CMSCST	44
	CM_CMSCSU	45
	CM_CMSCT	46
	CM_CMSCU	47
	CM_CMSDT	48
	CM_CMSED	49
	CM_CMSEND	50
	CM_CMSERR	51
	CM_CMSF	52
	CM_CMSID	53
	CM_CMSLD	54
	CM_CMSLTP	55
	CM_CMSMN	56
	CM_CMSNDX	57
	CM_CMSPDP	58
	CM_CMSPLN	60
	CM_CMSPM	61
	CM_CMSPTR	62
	CM_CMSQCF	63
	CM_CMSQPM	64
	CM_CMSRC	65

Variable or CharacteristicName	Pseudonym Values	Integer Values
	CM_CMSRT	66
	CM_CMSSL	67
	CM_CMSSRM	68
	CM_CMSST	69
	CM_CMSTC	70
	CM_CMSTPN	71
	CM_CMTRTS	72
	CM_CMWAIT	73
	CM_CMWCMP	74
	CM_CMSJT	75
confirmation_urgency	CM_CONFIRMATION_NOT_URGENT	0
	CM_CONFIRMATION_URGENT	1
control_information_received	CM_NO_CONTROL_INFO_RECEIVED	0
	CM_REQ_TO_SEND_RECEIVED	1
	CM_ALLOCATE_CONFIRMED	2
	CM_ALLOCATE_CONFIRMED_WITH_DATA	3
	CM_ALLOCATE_REJECTED_WITH_DATA	4
	CM_EXPEDITED_DATA_AVAILABLE	5
	CM_RTS_RCVD_AND_EXP_DATA_AVAIL	6
conversation_queue	CM_INITIALIZATION_QUEUE	0
	CM_SEND_QUEUE	1
	CM_RECEIVE_QUEUE	2
	CM_SEND_RECEIVE_QUEUE	3
	CM_EXPEDITED_SEND_QUEUE	4
	CM_EXPEDITED_RECEIVE_QUEUE	5
conversation_return_code	See *return_code*.	
conversation_security_type	CM_SECURITY_NONE	0
	CM_SECURITY_SAME	1
	CM_SECURITY_PROGRAM	2
	CM_SECURITY_PROGRAM_STRONG	5
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

Variable or CharacteristicName	Pseudonym Values	Integer Values
	CM_SYNC_POINT_SEND_STATE	12
	CM_SYNC_POINT_DEALLOCATE_STATE	13
	CM_INITIALIZE_INCOMING_STATE	14
	CM_SEND_ONLY_STATE	15
	CM_RECEIVE_ONLY_STATE	16
	CM_SEND_RECEIVE_STATE	17
	CM_PREPARED_STATE	18
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
expedited_receive_type	CM_RECEIVE_AND_WAIT	0
	CM_RECEIVE_IMMEDIATE	1
fill	CM_FILL_LL	0
	CM_FILL_BUFFER	1
join_transaction	CM_JOIN_IMPLICIT	0
	CM_JOIN_EXPLICIT	1
prepare_data_permitted	CM_PREPARE_DATA_NOT_PERMITTED	0
	CM_PREPARE_DATA_PERMITTED	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
queue_processing_mode	CM_BLOCKING	0
	CM_NON_BLOCKING	1

Variable or CharacteristicName	Pseudonym Values	Integer Values
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
and *conversation_return_code*	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_SYS	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_CONVERSATION_ENDING	38
	CM_SEND_RCV_MODE_NOT_SUPPORTED	39
	CM_BUFFER_TOO_SMALL	40
	CM_EXP_DATA_NOT_SUPPORTED	41
	CM_DEALLOC_CONFIRM_REJECT	42
	CM_ALLOCATION_ERROR	43
	CM_RETRY_LIMIT_EXCEEDED	44
	CM_NO_SECONDARY_INFORMATION	45
	CM_SECURITY_NOT_SUPPORTED	46

Variable or CharacteristicName	Pseudonym Values	Integer Values
	CM_CALL_NOT_SUPPORTED	48
	CM_PARM_VALUE_NOT_SUPPORTED	49
	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
	CM_CONV_DEALLOC_AFTER_SYNCPT	136
	CM_INCLUDE_PARTNER_REJECT_BO	137
return_control	CM_WHEN_SESSION_ALLOCATED	0
	CM_IMMEDIATE	1
send_receive_mode	CM_HALF_DUPLEX	0
	CM_FULL_DUPLEX	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_TAKE_COMMIT_DATA_OK	8
	CM_TAKE_COMMIT_SEND_DATA_OK	9
	CM_TAKE_COMMIT_DEALLOC_DATA_OK	10
	CM_PREPARE_OK	11
	CM_JOIN_TRANSACTION	12
sync_level	CM_NONE	0
	CM_CONFIRM	1
	CM_SYNC_POINT	2
	CM_SYNC_POINT_NO_CONFIRM	3
transaction_control	CM_CHAINED_TRANSACTIONS	0
	CM_UNCHAINED_TRANSACTIONS	1

Notes:

† The *call_ID* values greater than 10000 are reserved for the product extension calls.

‡ Early versions of CPI-C used the *request_to_send_received* variable. CPI-C 2.0 programs use *control_information_received*, which is an enhanced version of the *request_to_send_received* variable.

A.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 upper-case letters A through Z and numerals 0-9.
- Character set 00640, which is composed of the upper-case and lower-case letters A through Z, numerals 0-9, and 20 special characters.
- Character set T61String, which is composed of the upper-case and lower-case letters A through Z, numerals 0-9, and many additional special characters. The most commonly used special characters are provided in Table A-2. See the referenced CCITT Recommendation T.61 for other defined special characters.

These character sets, along with EBCDIC hexadecimal and graphic representations, are provided in Table A-2. See the referenced SNA Formats specification for more information on character sets.

Table A-2 Character Sets T61String, 01134 and 00640

EBCDIC Hex Code	Graphic	Description	Character Set		
			T61-String	01134	00640
40		Blank	X		X
4A	[	Left square bracket	X		
4B	.	Period	X		X
4C	<	Less than sign	X		X
4D	(	Left parenthesis	X		X
4E	+	Plus sign	X		X
4F	!	Exclamation mark	X		
50	&	Ampersand	X		X
5A	]	Right square bracket	X		
5C	*	Asterisk	X		X
5D	)	Right parenthesis	X		X
5E	;	Semicolon	X		X
60	-	Dash	X		X
61	/	Slash	X		X
6B	,	Comma	X		X
6C	%	Percent sign	X		X
6D	_	Underscore	X		X
6E	>	Greater than sign	X		X
6F	?	Question mark	X		X
7A	:	Colon	X		X
7C	@	Commercial a (at sign)	X		
7D	´	Single quote	X		X
7E	=	Equal sign	X		X
7F	"	Double quote	X		X
81	a	Lower-case a	X		X

EBCDIC Hex Code	Graphic	Description	Character Set		
			T61-String	01134	00640
82	b	Lower-case b	X		X
83	c	Lower-case c	X		X
84	d	Lower-case d	X		X
85	e	Lower-case e	X		X
86	f	Lower-case f	X		X
87	g	Lower-case g	X		X
88	h	Lower-case h	X		X
89	i	Lower-case i	X		X
91	j	Lower-case j	X		X
92	k	Lower-case k	X		X
93	l	Lower-case l	X		X
94	m	Lower-case m	X		X
95	n	Lower-case n	X		X
96	o	Lower-case o	X		X
97	p	Lower-case p	X		X
98	q	Lower-case q	X		X
99	r	Lower-case r	X		X
A2	s	Lower-case s	X		X
A3	t	Lower-case t	X		X
A4	u	Lower-case u	X		X
A5	v	Lower-case v	X		X
A6	w	Lower-case w	X		X
A7	x	Lower-case x	X		X
A8	y	Lower-case y	X		X
A9	z	Lower-case z	X		X
BB	|	Vertical line	X		
C1	A	Upper-case A	X	X	X
C2	B	Upper-case B	X	X	X
C3	C	Upper-case C	X	X	X
C4	D	Upper-case D	X	X	X
C5	E	Upper-case E	X	X	X
C6	F	Upper-case F	X	X	X
C7	G	Upper-case G	X	X	X
C8	H	Upper-case H	X	X	X
C9	I	Upper-case I	X	X	X
D1	J	Upper-case J	X	X	X
D2	K	Upper-case K	X	X	X
D3	L	Upper-case L	X	X	X
D4	M	Upper-case M	X	X	X
D5	N	Upper-case N	X	X	X
D6	O	Upper-case O	X	X	X
D7	P	Upper-case P	X	X	X
D8	Q	Upper-case Q	X	X	X
D9	R	Upper-case R	X	X	X
E2	S	Upper-case S	X	X	X

EBCDIC Hex Code	Graphic	Description	Character Set		
			T61-String	01134	00640
E3	T	Upper-case T	X	X	X
E4	U	Upper-case U	X	X	X
E5	V	Upper-case V	X	X	X
E6	W	Upper-case W	X	X	X
E7	X	Upper-case X	X	X	X
E8	Y	Upper-case Y	X	X	X
E9	Z	Upper-case Z	X	X	X
F0	0	Zero	X	X	X
F1	1	One	X	X	X
F2	2	Two	X	X	X
F3	3	Three	X	X	X
F4	4	Four	X	X	X
F5	5	Five	X	X	X
F6	6	Six	X	X	X
F7	7	Seven	X	X	X
F8	8	Eight	X	X	X
F9	9	Nine	X	X	X

A.3 Variable Types

CPI Communications makes use of two variable types, integer and character string. Table A-3 on page 341 defines the type and length of variables used in this document. Variable types are described below.

A.3.1 Integers

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

A.3.2 Character Strings

Character strings are composed of characters taken from one of the character sets discussed in Section A.2 on page 337, 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 specification 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 at the end of Table A-3 on page 341).

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 the notes following Table A-3 on page 341 (see the note regarding *partner_LU_name*), the concatenated character string as a whole is left-justified within the variable and the variable is filled out to the right with space characters. Space characters, if present, are not part of the concatenated character string.

Table A-3 Variable Types and Lengths

Variable	Variable Type	Character Set	Length
AE_qualifier [3, 4]	Character string	T61String	0-1024 bytes
AE_qualifier_format	Integer	N / A	32 bits
AE_qualifier_length	Integer	N / A	32 bits
allocate_confirm	Integer	N / A	32 bits
AP_title [3, 4]	Character string	T61String	0-1024 bytes
AP_title_format	Integer	N / A	32 bits
AP_title_length	Integer	N / A	32 bits
application_context_name [3, 4]	Character string	00640	0-256 bytes
application_context_name_length	Integer	N / A	32 bits
begin_transaction	Integer	N / A	32 bits
buffer [1, 2]	Character string	no restriction	0-max supported by system
buffer_length [1]	Integer	N / A	32 bits
call_ID	Integer	N / A	32 bits
callback_function	Pointer [10]	N / A	system-dependent [10]
completed_op_index_list	Array of integers	N / A	n X 32 bits
completed_op_count	Integer	N / A	32 bits
confirmation_urgency	Integer	N / A	32 bits
control_information_received	Integer	N / A	32 bits
conversation_ID	Character string	no restriction	8 bytes
conversation_queue	Integer	N / A	32 bits
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
expedited_receive_type	Integer	N / A	32 bits
fill	Integer	N / A	32 bits
initialization_data [3, 4]	Character string	no restriction	0-10000 bytes
initialization_data_length	Integer	N / A	32 bits
join_transaction	Integer	N / A	32 bits
log_data [3]	Character string	no restriction	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
OOID	Integer	N / A	32 bits
OOID_list	Array of integers	N / A	n X 32 bits
OOID_list_count	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_data_permitted	Integer	N / A	32 bits

Variable	Variable Type	Character Set	Length
prepare_to_receive_type	Integer	N / A	32 bits
processing_mode	Integer	N / A	32 bits
queue_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 [9]	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_receive_mode	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
timeout	Integer	N / A	32 bits
TP_name [3, 4, 6]	Character string	T61String	1-64 bytes
TP_name_length	Integer	N / A	32 bits
transaction_control	Integer	N / A	32 bits
user_field	Character string	no restriction	8 bytes
user_field_list	Array of character strings	no restriction	n X 8 bytes

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 the correct format (the negotiated transfer syntax on OSI TP and EBCDIC on LU 6.2) when they are used as input parameters on CPI Communications calls. 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 Section 3.8.5 on page 38 for more information on automatic conversion of these fields.

Note that an LU 6.2 CRM converts log data in character set 00640 only. To enhance program portability, it is recommended that character set 00640 be used for the *log_data* characteristic.

4. Because the *mode_name*, *partner_LU_name*, *security_user_ID*, *AE_qualifier*, *AP_title*, *application_context_name*, *TP_name*, and *initialization_data* 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 that an OSI TP CRM uses character set T61 String for the *AE_qualifier*, *AP_title*, *application_context_name*, and *TP_name*. An LU 6.2 CRM uses character set 00640 for the *partner_LU_name* and *TP_name*. Both CRM types use character set 01134 for the *mode_name*. To enhance program portability, it is recommended that character set 01134, a subset of character sets 00640 and T61 String, be used for these characteristics.

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 Section D.3 on page 480 and Section D.3.3 on page 481 for details.

7. The field containing the *sym_dest_name* parameter on the CMINIT call must be 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 195 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

9. The *request_to_send_received* variable is used by CPI-C 1.2 programs. CPI-C 2.0 programs use *control_information_received*, which is an enhanced version of this variable.

10. *callback_function* specifies a pointer to a routine and is supported by the C programming language only. Its length depends on the C compiler.

Appendix B

Return Codes and Secondary Information

This chapter discusses the parameter called *return_code* that is passed back to the program at the completion of a call. It also discusses associated secondary information that may be available for the program to extract using the Extract_Secondary_Information call.

B.1 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 or CM_SYNC_POINT_NO_CONFIRM, 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 5 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 A-1 on page 330 of Appendix A.

The valid *return_code* values are described below:

CM_ALLOCATE_FAILURE_NO_RETRY

The conversation cannot be allocated on a logical connection 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, if the conversation is using an LU 6.2 CRM, the logical connection, i.e., 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 logical connection 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 logical connection 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 logical connection is deactivated because of logical connection outage before the conversation can be allocated. The program can retry the allocation request. This return code is returned on the Allocate call.

CM_ALLOCATION_ERROR
: This may be returned on calls associated with the Send queue (except the Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND) while the conversation is in **Send-Receive** state. The function requested on the call is not performed.

  The return code indicates that the partner system rejected the conversation startup request. At the time this return code information is returned, the cause of allocation rejection is not returned to the program. The cause of the allocation rejection, which can be one of the following, can be obtained through the return code on the first Receive call:

  CM_SEND_RCV_MODE_NOT_SUPPORTED (OSI TP CRM only)
  CM_CONVERSATION_TYPE_MISMATCH (LU 6.2 CRM only)
  CM_PIP_NOT_SPECIFIED_CORRECTLY (LU 6.2 CRM only)
  CM_SECURITY_NOT_VALID (LU 6.2 CRM only)
  CM_SYNC_LVL_NOT_SUPPORTED_SYS (OSI TP CRM only)
  CM_SYNC_LVL_NOT_SUPPORTED_PGM (LU 6.2 CRM only)
  CM_TPN_NOT_RECOGNIZED
  CM_TP_NOT_AVAILABLE_NO_RETRY
  CM_TP_NOT_AVAILABLE_RETRY.

  The conversation is in **Receive-Only** state.

CM_BUFFER_TOO_SMALL
: The local program issued a CPI Communications call specifying a buffer size that is insufficient for the amount of data available for the program to receive. The state of the conversation remains unchanged.

CM_CALL_NOT_SUPPORTED
: The call is not supported by the local system. This return code is returned on any call in an optional support group when the implementation provides an entry point for the call but does not support the function requested by the call. The state of the conversation remains unchanged.

CM_CONV_DEALLOC_AFTER_SYNCPT (LU 6.2 CRM only)
: The conversation was deallocated as a part of the last sync-point operation. The local program was not given prior notification of the imminent deallocation because of a commit operation race that arose as follows: This program issued the resource recovery commit call. At the same time, the partner program issued a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT_NO_CONFIRM, followed by the commit call.

CM_CONVERSATION_ENDING (LU 6.2 CRM only)
This return code is returned on the Send_Expedited_Data and Receive_Expedited_Data calls and indicates one of the following:

— The local CRM is ending the conversation (normally).

— A notification indicating that the remote program is ending the conversation (normally or abnormally) has been received by the local CRM.

— A notification of an error that causes the conversation to terminate has been received from the remote CRM or occurred locally.

The error that causes the conversation to terminate may be an allocation error, a conversation failure, or a deallocation of the conversation. The return code indicating the cause of termination is returned on the calls associated with Send-Receive queue (half-duplex conversations only) or with the Send and Receive queues (full-duplex conversations only). The state of the conversation remains unchanged. Subsequent calls associated with the expedited queues will be rejected with this return code until the conversation enters **Reset** state.

CM_CONVERSATION_TYPE_MISMATCH (LU 6.2 CRM only)
The remote system rejected the conversation startup request because of one of the following:

— 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 conversation type.

— The local program issued an Allocate call with *send_receive_mode* set to either CM_HALF_DUPLEX or CM_FULL_DUPLEX, and the remote program does not support the respective send-receive mode.

For a half-duplex conversation, this return code is returned on a subsequent call to the Allocate. For a full-duplex conversation, this return code is returned on the Receive call. Calls associated with the Send queue that complete before this return code is returned on the Receive call are notified of the conversation type mismatch by a CM_ALLOCATION_ERROR return code.

When this *return_code* value is returned to the program, the conversation is in **Reset** state.

CM_DEALLOC_CONFIRM_REJECT (OSI TP CRM only)
This return code is returned on a full-duplex conversation under one of the following two conditions:

— The program issued a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_CONFIRM and the partner program responded negatively to the Deallocate by issuing a Send_Error call in **Confirm-Deallocate** state.

— The program issued a Send_Data call with *send_type* set to CM_SEND_AND_DEALLOCATE and *deallocate_type* set to CM_DEALLOCATE_CONFIRM, and the partner program responded negatively to the Send-Data by issuing a Send_Error call in **Confirm-Deallocate** state.

A Receive call issued by the local program receives a CM_PROGRAM_ERROR_PURGING return code after all available data is received. The state of the conversation remains unchanged.

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 a Cancel_Conversation call, or the remote system has done so because of a remote program abnormal-ending condition. If the remote program was in **Receive** state (half-duplex conversations only) or in **Send-Receive** or **Receive-Only** state (full-duplex conversations only) when the call was issued, information sent by the local program and not yet received by the remote program is purged.

— The remote 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 program.

— On a half-duplex conversation using an OSI TP CRM, the local program issued a Send_Error call, and the error notification was delivered to the remote CRM. Subsequently, the remote program issued a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_FLUSH, or with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* set to CM_NONE, or with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* set to CM_SYNC_POINT_NO_CONFIRM, and the conversation currently not included in a transaction.

— begin_transaction_collision (OSI TP CRM only)
On a full-duplex conversation, there was a collision between a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_CONFIRM issued by the local program and an Include_Partner_In_Transaction call issued by the partner program. No log data is available.

— dealloc_confirm_collision (OSI TP CRM only)
On a full-duplex conversation, there was a collision between a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_CONFIRM issued by the local program and a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_CONFIRM call issued by the partner program. No log data is available.

— On a full-duplex conversation in **Send-Receive** state or **Receive-Only** state, the local program has issued a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND.

— CPI Communications deallocated the conversation because an implicit call of *tx_set_transaction_control*( ) or *tx_begin*( ) failed.

For a half-duplex conversation, this return code is reported to the local program on a call issued in **Send** or **Receive** state. For a full-duplex conversation, this return code is returned on a Receive call issued in **Send-Receive** or **Receive-Only** state. It is also returned on calls associated with the Send queue (except the Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND) under one of the following conditions:

— They are issued in **Send-Only** state.

— They are issued in **Send-Receive** state and complete before this return code is returned on the Receive call.

The conversation is now in **Reset** state unless the return code was returned on one of the calls associated with the Send queue, issued in **Send-Receive** state. In that case, the conversation is in **Receive-Only** state.

CM_DEALLOCATED_ABEND_BO

This return code is returned only for conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and with the conversation included in a transaction.

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 a Cancel_Conversation call, or the remote system has done so because of a remote program abnormal-ending condition.

— dealloc_cfm_collision_bo (OSI TP CRM only)
On a full-duplex conversation, there was a collision between a Include_Partner_In_Transaction call issued by this program and a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_CONFIRM issued by the partner program. The Include_Partner_In_Transaction call got a return code of CM_OK. However, when the collision is detected, a CM_DEALLOCATED_ABEND_BO return code is returned on a subsequent call. No log data is available.

If the remote program was in **Receive** state (half-duplex conversations only) or in **Send-Receive**, **Prepared** or **Deferred-Deallocate** states (full-duplex conversations only) when it issued the Deallocate call, information sent by the local program and not yet received by the remote program is purged.

For a half-duplex conversation, this return code is reported to the local program on a call issued in **Send** or **Receive** state. For a full-duplex conversation, this return code is reported to the local program on calls issued in **Send-Receive**, **Sync-Point**, **Deferred-Deallocate**, **Sync-Point-Deallocate** and **Prepared** states. The conversation is now in **Reset** state. For a full-duplex conversation, incoming information may not be received if this return code is returned on a call associated with the Send queue.

The local program is in the **Backout-Required** condition, and the program must issue a resource recovery backout call in order to restore all protected resources to their status as of the last synchronization point.

CM_DEALLOCATED_ABEND_SVC (LU 6.2 CRM only)

This return code is returned for basic conversations only.

It may be returned under one of the following conditions:

— The remote 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 remote program was in **Receive** state (half-duplex conversations only) or in **Send-Receive** or **Receive-Only** state (full-duplex conversations only) when the verb was issued, information sent by the local program and not yet received by the remote program is purged.

— The remote 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 program.

For a half-duplex conversation, this return code is reported to the local program on a call issued in **Send** or **Receive** state. For a full-duplex conversation, this return code is returned on a Receive call issued in **Send-Receive** or **Receive-Only** state. It is also returned on calls associated with the Send queue (except the Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND) under one of the following conditions:

— They are issued in **Send-Only** state.

— They are issued in **Send-Receive** state and complete before this return code is returned on the Receive call.

The conversation is now in **Reset** state unless the return code was returned on one of the calls associated with the Send queue issued in **Send-Receive** state. In that case, the conversation is in **Receive-Only** state.

CM_DEALLOCATED_ABEND_SVC_BO (LU 6.2 CRM only)

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.

For a half-duplex conversation, this return code is reported to the local program on a call issued in **Send** or **Receive** state. For a full-duplex conversation, this return code is reported to the local program on calls issued in **Send-Receive**, **Sync-Point**, **Deferred-Deallocate**, **Sync-Point-Deallocate**, and **Prepared** states. The conversation is now in **Reset** state.

The local program is in the **Backout-Required** condition and the program must issue a resource recovery backout call in order to restore all protected resources to their status as of the last synchronization point.

CM_DEALLOCATED_ABEND_TIMER (LU 6.2 CRM only)

This return code is returned only for basic conversations.

In addition, it is returned only when the remote 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. For a half-duplex conversation, this return code is reported to the local program on a call issued in **Send** or **Receive** state. For a full-duplex conversation, this return code is returned on a Receive call issued in **Send-Receive** or **Receive-Only** state. It is also returned on calls associated with the Send queue (except the Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND) under one of the following conditions:

— They are issued in **Send-Only** state.

— They are issued in **Send-Receive** state and complete before this return code is returned on the Receive call.

The conversation is now in **Reset** state unless the return code was returned on one of the calls associated with the Send queue issued in **Send-Receive** state. In that case, the conversation is in **Receive-Only** state.

CM_DEALLOCATED_ABEND_TIMER_BO (LU 6.2 CRM only)

This return code is returned only for basic conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and only when the remote 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 (half-duplex conversations only) or in **Send-Receive**, **Prepared**, or **Deferred-Deallocate** state (full-duplex conversations only) when the verb was issued, information sent by the local program and not yet received by the remote program is purged. If the return code is returned on a call associated with the Send queue for a full-duplex conversation, incoming data may be purged. For a half-duplex conversation, this return

code is reported to the local program on a call issued in **Send** or **Receive** state. For a full-duplex conversation, this return code is reported to the local program on calls issued in **Send-Receive**, **Sync-Point**, **Deferred-Deallocate**, **Sync-Point-Deallocate**, and **Prepared** states. The conversation is now in **Reset** state.

The local program is in the **Backout-Required** condition and the program must issue a resource recovery backout call in order to restore all protected resources to their status as of the last synchronization point.

CM_DEALLOCATED_NORMAL

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

— The remote program issued a Deallocate call or a Send_Data call with *send_type* set to CM_SEND_AND_DEALLOCATE on a basic or mapped conversation with one of the following:

  — *deallocate_type* set to CM_DEALLOCATE_FLUSH

  — *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* set to CM_NONE

  — *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* set to CM_SYNC_POINT_NO_CONFIRM, and the conversation is not currently included in a transaction.

  For a half-duplex conversation, this return code is reported to the local program on a call issued in **Receive** state. For a full-duplex conversation, this return code is reported to the local program on the Receive call issued in **Send-Receive** or **Receive-Only** state. If the conversation is a full-duplex conversation using an OSI TP CRM, this return code is also returned on calls associated with the Send queue (except the Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND).

— The local program issued a Deallocate call or a Send_Data call with *send_type* set to CM_SEND_AND_DEALLOCATE and with one of the following:

  — *deallocate_type* set to CM_DEALLOCATE_FLUSH

  — *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* set to CM_NONE, and the conversation is a full-duplex conversation using an OSI TP CRM

  — *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* set to CM_SYNC_POINT_NO_CONFIRM, and the conversation is not currently included in a transaction.

  This return code is returned to the local program on a Receive call that was outstanding when the Deallocate call was issued.

For a half-duplex conversation, the conversation is now in **Reset** state. For a full-duplex conversation, the conversation can now be in one of the following states:

— **Reset** state if this return code was returned on calls issued in **Receive-Only** or **Send-Only** state.

— **Send-Only** state if this return code was returned on a Receive call issued in **Send-Receive** state.

— **Receive-Only** state if this return code was returned on calls associated with the Send queue (except the Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND) issued in **Send-Receive** state.

CM_DEALLOCATED_NORMAL_BO
This return code is returned only for half-duplex conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and with the conversation included in a transaction.

When the conversation is using an LU 6.2 CRM and 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 program is in the **Backout-Required** condition and the program must issue a resource recovery backout call in order to restore all protected resources to their status as of the last synchronization point.

CM_EXP_DATA_NOT_SUPPORTED (LU 6.2 CRM only)
An expedited data call was locally rejected because the remote CRM does not support expedited data. The state of the conversation remains unchanged.

CM_INCLUDE_PARTNER_REJECT_BO (OSI TP CRM only)
A prior Include_Partner_In_Transaction call issued by the program completed locally with CM_OK but was rejected by the partner system. The partner system rejected the request to join the transaction because the partner program is already a part of another transaction and cannot be a part of two transactions at the same time. The conversation is now in **Reset** state.

The local program is in the **Backout-Required** condition and the program must issue a resource recovery backout call in order to restore all protected resources to their status as of the last synchronization point.

CM_NO_SECONDARY_INFORMATION
The Extract_Secondary_Information call did not complete successfully because no secondary information was available for the specified call on the specified conversation. The state of the conversation remains unchanged.

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 either on the conversation (when conversation-level non-blocking is used) or on the queue with which the call is associated (when queue-level non-blocking is used), but the operation has not completed. This return code is returned when the call is suspended waiting for incoming data, buffers, or other resources. A program must do one of the following:

— For conversation-level non-blocking, use the Wait_For_Conversation call to wait for the operation to complete and to retrieve the return code for the completed operation.

— For queue-level non-blocking:

— If an OOID is associated with the outstanding operation, use the Wait_For_Completion call to wait for the operation to complete and to obtain the OOID and user field corresponding to the completed operation.

— If a callback function is associated with the outstanding operation, use the callback function, callback information, and user field to properly handle the completed operation.

The state of the conversation remains unchanged.

CM_OPERATION_NOT_ACCEPTED
A previous operation either on this conversation (when conversation-level non-blocking is chosen) or on the same queue (when conversation-level non-blocking is not chosen) is incomplete. This return code is returned when there is an outstanding operation on the conversation or queue, 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 is returned on a call made by another thread on the same conversation or associated with the same queue. The state of the conversation remains unchanged.

CM_PARM_VALUE_NOT_SUPPORTED
The specified value of a call parameter is not supported by the local system. This return code is returned on a call with defined parameter values that are optional for support of the call. It is returned when the implementation supports the call but does not support the specified optional parameter value. 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 (LU 6.2 CRM only)
This return code is returned only when the remote program is using an LU 6.2 application programming interface and is not using CPI Communications.

The remote CRM rejected the conversation startup request because the remote program has one or more program initialization parameter (PIP) variables defined and the initialization data specified by the local program is incorrect. This return code is returned on a call issued after the Allocate for a half-duplex conversation. For a full-duplex conversation, this return code is returned on the Receive call. Calls associated with the Send queue that complete before this return code is returned are notified of the error by an CM_ALLOCATION_ERROR return code. 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 (LU 6.2 CRM only)

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 (half-duplex conversations only) or in **Send-Receive** or **Send-Only** state (full-duplex conversations only). 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 (half-duplex conversations only) or in **Send-Receive** or **Send-Only** state (full-duplex conversations only), 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 half-duplex 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 for a half-duplex conversation or in **Send-Receive** or **Receive-Only** state for a full-duplex conversation.

CM_PROGRAM_ERROR_PURGING

One of the following occurred:

— The remote program issued a Send_Error call on a basic or mapped half-duplex 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 half-duplex 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 half-duplex 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 has issued a Send_Error call with *error_direction* set to CM_RECEIVE_ERROR.

— The full-duplex conversation is allocated using an OSI TP CRM and the remote program issued a Send_Error call while its end of the conversation was in **Send-Receive**, **Send-Only**, or **Confirm-Deallocate** state. Purging of data sent by the local program may have occurred in transit, if the remote program was in **Send-Receive** or **Send-Only** state when it issued Send_Error.

For a half-duplex conversation, 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. For a full-duplex conversation, this return code is returned on the Receive call. The half-duplex conversation remains in **Receive** state. The full-duplex conversation remains in **Send-Receive** or **Receive-Only** state.

CM_PROGRAM_ERROR_TRUNC (LU 6.2 CRM only)
The remote program issued a Send_Error call on a basic conversation, the conversation for the remote program was in **Send** state (half-duplex conversations only) or in **Send-Receive** or **Send-Only** state (full-duplex conversations only), 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 for a half-duplex conversation or in **Send-Receive** or **Receive-Only** state for a full-duplex conversation.

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 CRM type used by the conversation, and the transaction role (superior or subordinate) of the program.) 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.

— There is no incoming conversation. The Accept_Conversation call was issued but did not complete successfully.

— No name is associated with the program. The Accept_Conversation or Accept_Incoming call was issued but did not complete successfully.

— The program started but did not finish sending a logical record.

— There is no outstanding operation. The Wait_For_Completion or Wait_For_Conversation call was issued but did not complete successfully.

— For a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the program is in the **Backout-Required** condition. The program issued a call that is not allowed for this conversation while it is in this condition.

— The program has received a *status_received* value of CM_JOIN_TRANSACTION. The program issued a call that is not allowed before the program joins the transaction.

— The conversation is included in a transaction. The program issued a call that is allowed only when the conversation is not currently included in a transaction.

— The conversation is not currently included in a transaction. The program issued a call that is allowed only when the conversation is included in a transaction.

— A prior Deferred_Deallocate call is still in effect for the conversation. The Prepare_To_Receive call was issued but is not allowed.

— The program has not received a take-commit notification from its superior. The Prepare call was issued but is not allowed.

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 logical connection being used for the conversation was deactivated because of a logical-connection 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 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 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** state for a half-duplex or full-duplex conversation, or **Sync-Point**, **Sync-Point-Deallocate**, or **Defer-Deallocate** state for a full-duplex conversation.

For a full-duplex conversation, this return code is returned on the Receive call, at which time the conversation goes to **Reset** state. Calls associated with the Send queue (except the Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND) that complete before this return code is returned on the Receive call also get this return code, and the conversation is in **Receive-Only** or **Reset** state, depending on whether the call was issued in **Send-Receive** or **Send-Only** state. The conversation is in **Reset** state if this a half-duplex conversation.

CM_RESOURCE_FAIL_NO_RETRY_BO

This return code is returned only for conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and with the conversation included in a transaction.

A failure occurred that caused the conversation to be prematurely terminated. For example, the logical connection being used for the conversation was deactivated because of a logical-connection 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 issued in any state other than **Reset** or **Initialize** state. For a full-duplex conversation, incoming information may not be received if this return code is returned on a call associated with the Send queue. The conversation is in **Reset** state.

The local program is in the **Backout-Required** condition and the program must issue a resource recovery backout call in order to restore all protected resources to their status as of the last synchronization point.

CM_RESOURCE_FAILURE_RETRY

A failure occurred that caused the conversation to be prematurely terminated. For example, the logical connection being used for the conversation was deactivated because of a logical-connection 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** state for a half-duplex or full-duplex conversation, or **Sync-Point**, **Sync-Point-Deallocate**, or **Defer-Deallocate** state for a full-duplex conversation. For a full-duplex conversation, this return code is returned on the Receive call, at which time the conversation goes to **Reset** state. Calls associated with the Send queue (except the Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND) that complete before this return code is returned on the Receive call also get this return code, and the conversation is in **Receive-Only** or **Reset** state, depending on whether the call was issued in **Send-Receive** or **Send-Only** state. The conversation is in **Reset** state if this is a half-duplex conversation.

CM_RESOURCE_FAILURE_RETRY_BO

This return code is returned only for conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and with the conversation included in a transaction.

A failure occurred that caused the conversation to be prematurely terminated. For example, the logical connection being used for the conversation was deactivated because of a logical-connection 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**. For a full-duplex conversation, incoming information may not be received if this return code is returned on a call associated with the Send queue. The conversation is in **Reset** state.

The local program is in the **Backout-Required** condition and the program must issue a resource recovery backout call in order to restore all protected resources to their status as of the last synchronization point.

CM_RETRY_LIMIT_EXCEEDED

The conversation cannot be allocated on a logical connection because CPI Communications has exceeded the local system's retry limit. When this *return_code* value is returned to the program, the conversation is in **Reset** state.

CM_SECURITY_NOT_SUPPORTED

The local system rejected the allocate request because the local program specified a required user name type and conversation security type combination that is not supported between the local and remote systems. When this *return_code* value is returned to the program, the conversation is in **Reset** state.

CM_SECURITY_NOT_VALID

The remote system rejected the conversation startup request because the access security information (provided by the local system) is invalid. This return code is returned on a call issued after the Allocate for a half-duplex conversation. For a full-duplex conversation, this return code is returned on the Receive call. Calls associated with the Send queue that complete before this return code is returned on the Receive call are notified of the error by a CM_ALLOCATION_ERROR return code. When this *return_code* value is returned to the program, the conversation is in **Reset** state.

CM_SEND_RCV_MODE_NOT_SUPPORTED

This return code indicates that the conversation startup request was rejected because of one of the following:

— The *send_receive_mode* characteristic is set to CM_HALF_DUPLEX but the remote system does not support half-duplex conversations.

— The *send_receive_mode* characteristic is set to CM_FULL_DUPLEX but the remote system does not support full-duplex conversations.

The state of the conversation remains unchanged.

CM_SVC_ERROR_NO_TRUNC (LU 6.2 CRM only)

This return code is returned only for basic conversations. In addition, it is returned only when the remote 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 for a half-duplex conversation or in **Send-Receive** or **Send-Only** state for a full-duplex conversation, 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 on a half-duplex conversation, the conversation is in **Receive** state. There is no state change for a full-duplex conversation.

CM_SVC_ERROR_PURGING (LU 6.2 CRM only)

This return code is returned only for basic half-duplex conversations. In addition, it is returned only when the remote 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 (LU 6.2 CRM only)

This return code is returned only when the remote 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 for a half-duplex conversation or in **Send-Receive** or **Send-Only** state for a full-duplex conversation, 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 state of the conversation remains unchanged.

CM_SYNC_LVL_NOT_SUPPORTED_SYS
: This return code is returned only for conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.

  The local program specified a *sync_level* of CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, which the remote system does not support. This return code is returned on the Allocate call.

  For a full-duplex conversation, this return code is returned on the Receive call if an attempt to allocate the conversation was made by the local program running on an OSI TP CRM and the remote system does not support the *sync_level* of CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM specified in the conversation startup request. The remote system has rejected the allocation attempt. Calls associated with the local Send queue that complete before this return code is returned on the Receive call are notified of the error by a CM_ALLOCATION_ERROR return code.

  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 conversation startup request because the local program specified a synchronization level (with the *sync_level* parameter) that the remote program does not support. For a half-duplex conversation, this return code is returned on a call issued after the Allocate. For a full-duplex conversation, this return code is returned on the Receive call. Calls associated with the Send queue that complete before this return code is returned on the Receive call are notified of the error by a CM_ALLOCATION_ERROR return code.

  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 or CM_SYNC_POINT_NO_CONFIRM, and with the conversation included in a transaction.

  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 to their status as of the last synchronization point. The program 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 conversation startup request because the local program specified a remote program name that the remote system does not recognize. For a half-duplex conversation, this return code is returned on a call issued after the Allocate. For a full-duplex conversation, this return code is returned on the Receive call. Calls associated with the Send queue that complete before this return code is returned on the Receive call are

notified of the error by a CM_ALLOCATION_ERROR return code. 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 conversation startup 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. For a half-duplex conversation, this return code is returned on a call issued after the Allocate. For a full-duplex conversation, this return code is returned on the Receive call. Calls associated with the Send queue that complete before this return code is returned on the Receive call are notified of the error by a CM_ALLOCATION_ERROR return code. When this *return_code* value is returned to the program, the conversation is in **Reset** state.

CM_TP_NOT_AVAILABLE_RETRY
: The remote system rejected the conversation startup 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. For a half-duplex conversation, this return code is returned on a call issued after the Allocate. For a full-duplex conversation, this return code is returned on the Receive call. Calls associated with the Send queue that complete before this return code is returned on the Receive call are notified of the error by a CM_ALLOCATION_ERROR return code. 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.

B.2 Secondary Information

Associated with the return code, there may be secondary information available for the program to extract using the Extract_Secondary_Information call. The secondary information can be used to determine the cause of the return code and to aid problem determination. Based on its origin, the secondary information and associated return code can belong to one of the four types, as shown in Table B-1.

Table B-1 Secondary Information Types and Associated Return Codes

Secondary Information Type	Associated Return Codes
Application-oriented	CM_DEALLOCATED_ABEND CM_DEALLOCATED_ABEND_BO CM_DEALLOCATED_ABEND_SVC CM_DEALLOCATED_ABEND_SVC_BO CM_DEALLOCATED_ABEND_TIMER CM_DEALLOCATED_ABEND_TIMER_BO CM_PROGRAM_ERROR_NO_TRUNC CM_PROGRAM_ERROR_PURGING CM_PROGRAM_ERROR_TRUNC CM_SVC_ERROR_NO_TRUNC CM_SVC_ERROR_PURGING CM_SVC_ERROR_TRUNC
CPI Communications-defined	CM_DEALLOCATED_ABEND CM_DEALLOCATED_ABEND_BO CM_PARAMETER_ERROR CM_PROGRAM_PARAMETER_CHECK CM_PROGRAM_STATE_CHECK CM_SECURITY_NOT_SUPPORTED
CRM-specific	CM_ALLOCATE_FAILURE_NO_RETRY CM_ALLOCATE_FAILURE_RETRY CM_CONVERSATION_TYPE_MISMATCH CM_PIP_NOT_SPECIFIED_CORRECTLY CM_RESOURCE_FAIL_NO_RETRY_BO CM_RESOURCE_FAILURE_NO_RETRY CM_RESOURCE_FAILURE_RETRY CM_RESOURCE_FAILURE_RETRY_BO CM_RETRY_LIMIT_EXCEEDED CM_SECURITY_NOT_SUPPORTED CM_SECURITY_NOT_VALID CM_SEND_RCV_MODE_NOT_SUPPORTED CM_SYNC_LVL_NOT_SUPPORTED_PGM CM_SYNC_LVL_NOT_SUPPORTED_SYS CM_TP_NOT_AVAILABLE_NO_RETRY CM_TP_NOT_AVAILABLE_RETRY CM_TPN_NOT_RECOGNIZED
Implementation-related	CM_PRODUCT_SPECIFIC_ERROR

The following return codes, upon being returned to the program, are not associated with any secondary information:

CM_ALLOCATION_ERROR
CM_BUFFER_TOO_SMALL
CM_CALL_NOT_SUPPORTED
CM_CONV_DEALLOC_AFTER_SYNCPT
CM_CONVERSATION_ENDING
CM_DEALLOC_CONFIRM_REJECT
CM_DEALLOCATED_NORMAL
CM_DEALLOCATED_NORMAL_BO
CM_EXP_DATA_NOT_SUPPORTED
CM_INCLUDE_PARTNER_REJECT_BO
CM_NO_SECONDARY_INFORMATION
CM_OK
CM_OPERATION_INCOMPLETE
CM_OPERATION_NOT_ACCEPTED
CM_PARM_VALUE_NOT_SUPPORTED
CM_SYSTEM_EVENT
CM_TAKE_BACKOUT
CM_UNSUCCESSFUL.

Except for application-oriented information, which is defined entirely by the application, secondary information is a string of printable characters and, in general, consists of the following information in the order listed:

1. Condition code
2. Description of the condition
3. Cause of the condition
4. Suggested actions
5. Additional information from the implementation.

For different secondary information types, the condition codes are in the range specified in Table B-2. In some cases, secondary information may not have all these fields. Fields present in secondary information are separated by two consecutive semicolons. The following sections provide examples of secondary information in different types.

Table B-2 Range of Condition Codes for Different Secondary Information Types

Secondary Information Type	Condition Codes
CPI Communications-defined	1 - 4000
CRM-specific	4001 (for an LU 6.2 CRM) 4002 (for an OSI TP CRM)
Implementation-related	4003

B.2.1 Application-oriented Information

When a program discovers an abnormal condition during its processing, the program may use log data to convey the condition to its partner program. The partner program receives the log data when it issues the Extract_Secondary_Information call. Since log data is application data, it is up to the application designer to define and interpret its content.

B.2.2 CPI Communications-defined Information

Table B-3 lists all CPI Communications-defined secondary information.

Table B-3 CPI Communications-defined Secondary Information

Condition Code	Description
Associated with CM_PROGRAM_PARAMETER_CHECK:	
0 < *n* < 101	The *n*th parameter specifies an invalid address.
101	The *conversation_ID* specifies an unassigned conversation identifier.
102	The *sync_level* is set to CM_NONE.
103	The *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM.
104	The *send_receive_mode* is set to CM_FULL_DUPLEX.
105	The *send_receive_mode* is set to CM_FULL_DUPLEX, and conversation is using an LU 6.2 CRM.
106	The *buffer_length* specifies a value that is invalid for the range permitted by the implementation.
107	The conversation is using an OSI TP CRM, and the program is not the superior for the conversation.
108	The conversation is using an LU 6.2 CRM.
109	The *requested_length* specifies a value less than 0.
110	The *transaction_control* is set to CM_CHAINED_TRANSACTIONS.
111	The *sym_dest_name* specifies an unrecognized value.
112	The *sync_level* is set to CM_CONFIRM.
113	The *requested_length* specifies a value that exceeds the range permitted by the implementation.
114	The *requested_length* specifies a value less than 0 or greater than 86.
115	The *expedited_receive_type* specifies an undefined value.

Condition Code	Description
116	The conversation is using an OSI TP CRM.
117	The *TP_name* specifies a name that is not associated with this program.
118	The *TP_name_length* specifies a value less than 1 or greater than 64.
119	The *send_length* specifies a value that exceeds the range permitted by the implementation.
120	The *conversation_type* is set to CM_BASIC_CONVERSATION and *buffer* contains an invalid logical record length (LL) value of X'0000', X'0001', X'8000', X'8001'.
121	The *send_type* is set to CM_SEND_AND_PREP_TO_RECEIVE, *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT, the conversation using an OSI TP CRM is included in a transaction, and the program is not the superior for the conversation.
122	The *send_type* is set to CM_SEND_AND_PREP_TO_RECEIVE, *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, the conversation using an OSI TP CRM is included in a transaction, and the program is not the superior for the conversation.
123	The *send_type* is set to CM_SEND_AND_DEALLOCATE, *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT, the conversation using an OSI TP CRM is included in a transaction, and the program is not the superior for the conversation.
124	The *send_type* is set to CM_SEND_AND_DEALLOCATE, *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, the conversation using an OSI TP CRM is included in a transaction, and the program is not the superior for the conversation.
125	The *send_length* specifies a value less than 1 or greater than 86.
126	The *AE_qualifier_length* specifies a value less than 1 or greater than 1024.
127	The *AE_qualifier_format* specifies an undefined value.
128	Reserved, in use by the CPI-C Implementers' Workshop (CIW).
129	The *allocate_confirm* specifies an undefined value.
130	The *allocate_confirm* specifies CM_ALLOCATE_CONFIRM, and the conversation is using an LU 6.2 CRM.
131	The *AP_title_length* specifies a value less than 1 or greater than 1024.
132	The *AP_title_format* specifies an undefined value.

Condition Code	Description
133	The *application_context_name_length* specifies a value less than 1 or greater than 256.
134	The *begin_transaction* specifies an undefined value.
135	The *confirmation_urgency* specifies an undefined value.
136	The *security_password_length* specifies a value less than 0 or greater than 10.
137	The *conversation_security_type* specifies an undefined value.
138	The *security_user_ID_length* specifies a value less than 0 or greater than 10.
139	The *conversation_type* specifies an undefined value.
140	The *conversation_type* specifies CM_MAPPED_CONVERSATION, and *fill* is set to CM_FILL_BUFFER.
141	The *conversation_type* specifies CM_MAPPED_CONVERSATION, and a prior call to Set_Log_Data is still in effect.
142	The *deallocate_type* specifies CM_DEALLOCATE_FLUSH, *sync_level* is set to CM_SYNC_POINT, and *transaction_control* is set to CM_CHAINED_TRANSACTIONS.
143	The *deallocate_type* specifies CM_DEALLOCATE_FLUSH, *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, and *transaction_control* is set to CM_CHAINED_TRANSACTIONS.
144	The *deallocate_type* specifies CM_DEALLOCATE_CONFIRM, *sync_level* is set to CM_NONE, and the conversation is using an LU 6.2 CRM.
145	The *deallocate_type* specifies CM_DEALLOCATE_CONFIRM, *sync_level* is set to CM_SYNC_POINT, and the conversation is using an LU 6.2 CRM.
146	The *deallocate_type* specifies CM_DEALLOCATE_CONFIRM, *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, and the conversation is using an LU 6.2 CRM.
147	The *deallocate_type* specifies CM_DEALLOCATE_CONFIRM, *sync_level* is set to CM_SYNC_POINT, *transaction_control* is set to CM_CHAINED_TRANSACTIONS, and the conversation is using an OSI TP CRM.
148	The *deallocate_type* specifies CM_DEALLOCATE_CONFIRM, *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, *transaction_control* is set to CM_CHAINED_TRANSACTIONS, and the conversation is using an OSI TP CRM.
149	The *deallocate_type* specifies an undefined value.

Condition Code	Description
150	The *error_direction* specifies CM_SEND_ERROR, and the conversation is using an OSI TP CRM.
151	The *error_direction* specifies an undefined value.
152	The *conversation_type* is set to CM_MAPPED_CONVERSATION.
153	The *fill* specifies an undefined value.
154	The *initialization_data_length* specifies a value less than 0 or greater than 10000.
155	The *conversation_type* is set to CM_MAPPED_CONVERSATION, and the conversation is using an LU 6.2 CRM.
156	The *log_data_length* specifies a value less than 0 or greater than 512.
157	Reserved for future use by the CPI-C Implementers' Workshop (CIW).
158	Reserved for future use by the CPI-C Implementers' Workshop (CIW).
159	Reserved for future use by the CPI-C Implementers' Workshop (CIW).
160	Reserved for future use by the CPI-C Implementers' Workshop (CIW).
161	Reserved for future use by the CPI-C Implementers' Workshop (CIW).
162	Reserved for future use by the CPI-C Implementers' Workshop (CIW).
163	Reserved for future use by the CPI-C Implementers' Workshop (CIW).
164	The *partner_LU_name_length* specifies a value less than 1 or greater than 17.
165	The *prepare_data_permitted* specifies CM_PREPARE_DATA_PERMITTED, and the conversation is using an LU 6.2 CRM.
166	The *prepare_data_permitted* specifies an undefined value.
167	The *prepare_to_receive_type* specifies CM_PREP_TO_RECEIVE_CONFIRM, and *sync_level* set to CM_NONE.
168	The *prepare_to_receive_type* specifies CM_PREP_TO_RECEIVE_CONFIRM, and *sync_level* set to CM_SYNC_POINT_NO_CONFIRM.
169	The *prepare_to_receive_type* specifies an undefined value.
170	The *processing_mode* specifies an undefined value.
171	The program has chosen queue-level non-blocking for the conversation.
172	The *conversation_queue* specifies a value that is not defined for the *send_receive_mode* conversation characteristic.

Condition Code	Description
173	The program has chosen conversation-level non-blocking for the conversation.
174	The *queue_processing_mode* specifies an undefined value.
175	The *receive_type* specifies an undefined value.
176	The *return_control* specifies an undefined value.
177	The *send_receive_mode* specifies CM_FULL_DUPLEX, and *sync_level* is set to CM_CONFIRM.
178	The *send_receive_mode* specifies CM_FULL_DUPLEX, and *sync_level* is set to CM_SYNC_POINT.
179	The *send_receive_mode* specifies CM_FULL_DUPLEX, and *send_type* is set to CM_SEND_AND_PREP_TO_RECEIVE.
180	The *send_receive_mode* specifies CM_FULL_DUPLEX, and the program has chosen conversation-level non-blocking for the conversation.
181	The *send_receive_mode* specifies an undefined value.
182	The *send_type* specifies CM_SEND_AND_CONFIRM, and *sync_level* is set to CM_NONE.
183	The *send_type* specifies CM_SEND_AND_CONFIRM, and *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM.
184	The *send_type* specifies CM_SEND_AND_CONFIRM, and *send_receive_mode* is set to CM_FULL_DUPLEX.
185	The *send_type* specifies CM_SEND_AND_PREP_TO_RECEIVE, and *send_receive_mode* is set to CM_FULL_DUPLEX.
186	The *send_type* specifies an undefined value.
187	The *sync_level* specifies CM_NONE, *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, and the conversation is using an LU 6.2 CRM.
188	The *sync_level* specifies CM_NONE, *send_receive_mode* is set to CM_HALF_DUPLEX, and *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_CONFIRM.
189	The *sync_level* specifies CM_NONE, *send_receive_mode* is set to CM_HALF_DUPLEX, and *send_type* is set to CM_SEND_AND_CONFIRM.
190	The *sync_level* specifies CM_SYNC_POINT_NO_CONFIRM, *send_receive_mode* is set to CM_HALF_DUPLEX, and *send_type* is set to CM_SEND_AND_CONFIRM.

Condition Code	Description
191	The *sync_level* specifies CM_CONFIRM, and *send_receive_mode* is set to CM_FULL_DUPLEX.
192	The *sync_level* specifies CM_SYNC_POINT, and *send_receive_mode* is set to CM_FULL_DUPLEX.
193	The *sync_level* specifies CM_SYNC_POINT, *deallocate_type* is set to CM_DEALLOCATE_FLUSH, and the conversation is using an LU 6.2 CRM.
194	The *sync_level* specifies CM_SYNC_POINT, *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, and the conversation is using an LU 6.2 CRM.
195	The *sync_level* specifies CM_SYNC_POINT_NO_CONFIRM, *deallocate_type* is set to CM_DEALLOCATE_FLUSH, and the conversation is using an LU 6.2 CRM.
196	The *sync_level* specifies CM_SYNC_POINT_NO_CONFIRM, *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, and the conversation is using an LU 6.2 CRM.
197	The *sync_level* specifies CM_SYNC_POINT_NO_CONFIRM, *send_receive_mode* is set to CM_HALF_DUPLEX, and the conversation is using an LU 6.2 CRM.
198	The *sync_level* specifies an undefined value.
199	The *transaction_control* specifies CM_UNCHAINED_TRANSACTIONS, and the conversation is using an LU 6.2 CRM.
200	The *transaction_control* specifies an undefined value.
201	The *TP_name_length* specifies a value less than 1 or greater than 64.
202	The *TP_name* specifies a name that is restricted in some way by node services.
203	The *TP_name* has incorrect internal syntax as defined by node services.
204	The *TP_name_length* specifies a value less than 1 or greater than 64.
205	The *OOID_list_count* specifies a value less than 1.
206	The number of OOIDs in *OOID_list* is less than the value specified in *OOID_list_count*.
207	The *OOID_list* contains an unassigned OOID.
208	The *timeout* specifies a value less than 0.
209	The *join_transaction* specifies an undefined value.
210	The program is not the subordinate for the conversation.

Condition Code	Description
211	The *transaction_control* is set to CM_CHAINED_TRANSACTIONS.
212-1000	Reserved for future conditions to be associated with CM_PROGRAM_PARAMETER_CHECK.

Condition Code	Description
Associated with CM_PROGRAM_STATE_CHECK:	
1001	No incoming conversation exists.
1002	No name is associated with the program. A program associates a name with itself by issuing the Specify_Local_TP_Name call.
1003	The conversation is not in **Initialize-Incoming** state.
1004	The conversation is not in **Initialize** state.
1005	The program is in the **Backout-Required** condition.
1006	The conversation is not in **Send**, **Send-Pending**, or **Defer-Receive** state.
1007	The conversation is basic, and the program started but did not finish sending a logical record.
1008	The conversation is not in **Confirm**, **Confirm-Send**, or **Confirm-Deallocate** state.
1009	The conversation is not in **Confirm-Deallocate** state.
1010	The conversation is not in **Send** or **Send-Pending** state.
1011	The *deallocate_type* is set to CM_DEALLOCATE_FLUSH, and the conversation is currently included in a transaction.
1012	The *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, and the conversation is currently included in a transaction.
1013	The program has received a *status_received* value of CM_JOIN_TRANSACTION and must issue a *tx_begin*( ) call to the TX (Transaction Demarcation) interface to join the transaction.
1014	The conversation is not in **Send-Receive** or **Send-Only** state.
1015	The conversation is not in **Send-Receive** state.
1016	The conversation is not currently included in a transaction.
1017	The conversation is in **Initialize-Incoming** state.
1018	The conversation is in **Initialize** state.
1019	The conversation is not included in a transaction. The program must issue a *tx_begin*( ) call to the TX (Transaction Demarcation) interface to start a transaction.
1020	The conversation is already included in the current transaction.

Condition Code	Description
1021	The conversation is using an OSI TP CRM, *begin_transaction* is set to CM_BEGIN_EXPLICIT, and the conversation is not currently included in a transaction.
1022	The conversation is using an OSI TP CRM, and the program is not the root of the transaction and has not received a take-commit notification from its superior.
1023	A prior call to Deferred_Deallocate is still in effect for the conversation.
1024	The *receive_type* is set to CM_RECEIVE_AND_WAIT, and the conversation is not in **Send**, **Receive**, **Send-Pending**, or **Prepared** state.
1025	The *receive_type* is set to CM_RECEIVE_IMMEDIATE, and the conversation is not in **Receive** or **Prepared** state.
1026	The conversation is not in **Send-Receive**, **Receive-Only**, or **Prepared** state.
1027	The conversation is not in **Send**, **Receive**, **Send-Pending**, **Confirm**, **Confirm-Send**, **Confirm-Deallocate**, **Sync-Point**, **Sync-Point-Send**, **Sync-Point-Deallocate**, or **Prepared** state.
1028	For a conversation using an OSI TP CRM, the Request_To_Send call is not allowed from **Send** state.
1029	The conversation is not in **Send**, **Send-Pending**, **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** state.
1030	The program received a take-commit notification not ending in *_DATA_OK, and the conversation is in **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** state.
1031	The *send_type* is set to CM_SEND_AND_CONFIRM or CM_SEND_AND_PREP_TO_RECEIVE, and the conversation is in **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** state.
1032	The *send_type* is set to CM_SEND_AND_DEALLOCATE, *deallocate_type* is not set to CM_DEALLOCATE_ABEND, and the conversation is in **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** state.
1033	The *send_type* is set to CM_SEND_AND_DEALLOCATE, *deallocate_type* is set to CM_DEALLOCATE_FLUSH, *sync_level* is set to CM_SYNC_POINT, and the conversation is included in a transaction.
1034	The *send_type* is set to CM_SEND_AND_DEALLOCATE, *deallocate_type* is set to CM_DEALLOCATE_FLUSH, *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, and the conversation is included in a transaction.
1035	The *send_type* is set to CM_SEND_AND_DEALLOCATE, *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, *sync_level* is set to CM_SYNC_POINT, and the conversation is included in a transaction.

Condition Code	Description
1036	The *send_type* is set to CM_SEND_AND_DEALLOCATE, *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, and the conversation is included in a transaction.
1037	The conversation is not in **Send-Receive**, **Send-Only**, **Sync-Point**, or **Sync-Point-Deallocate** state.
1038	The program receives a take-commit notification not ending in *_DATA_OK, and the conversation is in **Sync-Point** or **Sync-Point-Deallocate** state.
1039	The *send_type* is set to CM_SEND_AND_DEALLOCATE, *deallocate_type* is not set to CM_DEALLOCATE_ABEND, and the conversation is in **Sync-Point** or **Sync-Point-Deallocate** state.
1040	The conversation is not in **Send-Receive**, **Send-Only**, or **Confirm-Deallocate** state.
1041	The *conversation_security_type* is not set to CM_SECURITY_PROGRAM or CM_SECURITY_PROGRAM_STRONG.
1042	The conversation is not in **Initialize** or **Initialize-Incoming** state.
1043	The conversation is not in **Initialize** or **Receive** state.
1044	The conversation is not in **Initialize** or **Send-Receive** state.
1045	The *conversation_queue* specifies CM_INITIALIZATION_QUEUE, and the conversation is not in **Initialize** or **Initialize-Incoming** state.
1046	The *conversation_queue* specifies a value other than CM_INITIALIZATION_QUEUE, and the conversation is in **Initialize-Incoming** state.
1047	The conversation is not in **Send**, **Receive**, **Send-Pending**, **Defer-Receive**, or **Defer-Deallocate** state.
1048	There is no outstanding operation associated with any of the OOIDs specified in *OOID_list* or by use of a defined value of *OOID_list_count* has completed.
1049	There were no conversation-level outstanding operations for the program.
1050	The *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, *transaction_control* is set to CM_CHAINED_TRANSACTIONS, and the program is not in transaction mode.
1051	The program has issued a successful Accept_Conversation (CMACCP) or Accept_Incoming (CMACCI) call on a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and using an OSI TP CRM, and the program has not issued a Receive (CMRCV) call on this conversation.

Condition Code	Description
1052	The *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, CM_DEALLOCATE_FLUSH or CM_DEALLOCATE_CONFIRM, the local program is the superior to the conversation, it has issued the Allocate (CMALLC) call with the *allocate_confirm* characteristic set to CM_ALLOCATE_CONFIRM, and it did not yet receive a *control_information_received* value of CM_ALLOCATE_CONFIRMED or CM_ALLOCATE_CONFIRMED_WITH_DATA.
1053	The Receive (CMRCV) call is the first activity on the conversation following Accept_Conversation (CMACCP) or Accept_Incoming (CMACCI), *join_transaction* is set to CM_JOIN_EXPLICIT, *transaction_control* is CM_CHAINED_TRANSACTION and the program has not issued a *tx_begin*( ) call to the TX (Transaction Demarcation) interface to join the transaction.
1054	The Send_Error (CMSERR) call is the first activity on the conversation following Accept_Conversation (CMACCP) or Accept_Incoming (CMACCI), *join_transaction* is set to CM_JOIN_EXPLICIT, *transaction_control* is CM_CHAINED_TRANSACTION and the program has not issued a *tx_begin*( ) call to the TX (Transaction Demarcation) interface to join the transaction.
1055-2000	Reserved for future conditions to be associated with CM_PROGRAM_STATE_CHECK.

Condition Code	Description
Associated with CM_PARAMETER_ERROR:	
2001	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.
2002	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.
2003	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.
2004	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.
2005	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.
2006	The *AP_title* characteristic (set from side information or using Set_AP_Title call or the *AE_qualifier* characteristic (set from side information or using Set_AE_Qualifier call) or the *application_context_name* characteristic (set from side information or using the Set_Application_Context_Name call) specifies an AP title or an AE qualifier or an application context name that is not recognized as being valid.
2007	The *conversation_security_type* characteristic is set to CM_SECURITY_PROGRAM or CM_SECURITY_PROGRAM_STRONG, and the *security_password* characteristic or the *security_user_ID* characteristic (set from side information or by Set calls) or both, are null.
2008-2500	Reserved for future conditions to be associated with CM_PARAMETER_ERROR.

Condition Code	Description
Associated with CM_SECURITY_NOT_SUPPORTED:	
2501	Reserved for future use by the CPI-C Implementers' Workshop (CIW).
2502-3000	Reserved for future conditions to be associated with CM_SECURITY_NOT_SUPPORTED.

Condition Code	Description
Associated with CM_DEALLOCATED_ABEND (full-duplex conversations using an OSI TP CRM only):	
3001	There was a collision between a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_CONFIRM issued by the local program and an Include_Partner_In_Transaction call issued by the partner program. No log data is available.
3002	There was a collision between a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_CONFIRM issued by the local program and a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_CONFIRM call issued by the partner program. No log data is available.
3003	CPI Communications deallocated the incoming conversation because an implicit call of *tx_set_transaction_control*( ) failed with TX return code [TX_PROTOCOL_ERROR].
3004	CPI Communications deallocated the incoming conversation because an implicit call of *tx_set_transaction_control*( ) failed with TX return code [TX_FAIL].
3005	CPI Communications deallocated the conversation because an implicit call of *tx_begin*( ) failed with TX return code [TX_OUTSIDE].
3006	CPI Communications deallocated the conversation because an implicit call of *tx_begin*( ) failed with TX return code [TX_PROTOCOL_ERROR].
3007	CPI Communications deallocated the conversation because an implicit call of *tx_begin*( ) failed with TX return code [TX_ERROR].
3008	CPI Communications deallocated the conversation because an implicit call of *tx_begin*( ) failed with TX return code [TX_FAIL].
3009-3500	Reserved for future conditions to be associated with CM_DEALLOCATED_ABEND.

Condition Code	Description
Associated with CM_DEALLOCATED_ABEND_BO (full-duplex conversations using an OSI TP CRM only):	
3501	There was a collision between a Include_Partner_In_Transaction call issued by the local program and a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_CONFIRM issued by the partner program. No log data is available.
3502-4000	Reserved for future conditions to be associated with CM_DEALLOCATED_ABEND_BO.

B.2.3 CRM-specific Information

When the underlying CRM discovers an abnormal condition, the condition, identified by a CRM-specific message, is then mapped to a CPI Communications return code and returned to the program. The CRM-specific message is the OSI diagnostic information for the CRM type of OSI TP and SNA sense data information for the CRM type of LU 6.2.

Table B-4 Examples of Secondary Information from an OSI TP CRM

Associated with CM_RESOURCE_FAILURE_NO_RETRY, CM_RESOURCE_FAIL_NO_RETRY_BO:
4002;;recipient-unknown 4002;;no-reason-given 4002;;permanent-failure 4002;;protocol-error
Associated with CM_TPN_NOT_RECOGNIZED:
4002;;recipient-tpsu-title-unknown 4002;;recipient-tpsu-title-required
Associated with CM_SYNC_LVL_NOT_SUPPORTED_SYS:
4002;;functional-unit-not-supported 4002;;functional-unit-combination-not-supported

Table B-5 Examples of Secondary Information from an LU 6.2 CRM

Associated with CM_CONVERSATION_TYPE_MISMATCH:
4001;;1008 6034 The FMH-5 Attach command specifies a conversation type that the receiver does not support for the specified transaction program. This sense data is sent only in FMH-7.
Associated with CM_TPN_NOT_RECOGNIZED:
4001;;1008 6021 Transaction Program Name Not Recognized: The FMH-5 Attach command specifies a transaction program name that the receiver does not recognize. This sense data is sent only in FMH-7.
Associated with CM_SYNC_LVL_NOT_SUPPORTED_SYS:
4001;;1008 6040 Invalid Attach Parameter: A parameter in the FMH-5 Attach command conflicts with the statements of LU capability previously provided in the BIND negotiation.

Note: See Chapter 10 of System Network Architecture Formats (IBM document number GA27-3136) for complete information about sense data.

B.2.4 Implementation-related Information

An implementation may return CM_PRODUCT_SPECIFIC_ERROR to the program for any errors that are specific to the implementation or to the system that supports the implementation. In this case, secondary information is the error message defined by the implementation or system.

Table B-6 Examples of Implementation-related Secondary Information

Example 1:	
	4003;;0001 STACK_TOO_SMALL;; A minimum stack size of 3500 bytes is required by CPI-C when a call is issued. CPI-C runs on the stack of the program that calls it. When the call was issued, CPI-C found the stack size to be less than the minimum size.;;**Programmer Response:** Increase the stack size specified in the **.DEF** file used in linking. If your program calls CPI-C from a thread it has created, be sure the stack size on the NewThreadStack parameter of your DosCreateThread function call is large enough.
Example 2:	
	4003;;0002 CANNOT_ALLOCATE_SHARED_SEGMENT;; Communication Resource Manager could not allocate the shared segment named `\SHAREMEM\ACSLGMEM`.; ;**Programmer Response:** A necessary shared segment is not currently available. There is no corrective action that your program can take. This problem will recur until the Communication Resource Manager is stopped and restarted. **Operator Response:** Communication Resource Manager must be restarted to correct the problem.

Appendix C

State Tables

The CPI Communications state tables show 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 Section 3.13 on page 49, 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 C-1 on page 391 describes the state transitions that are allowed for the CPI Communications calls on half-duplex conversations. Table C-4 on page 409 describes the state transitions that are allowed for CPI Communications calls on full-duplex conversations.

Table C-2 on page 398 shows the effects of TX (Transaction Demarcation) calls on CPI Communications conversation states for half-duplex conversations. Table C-3 on page 400 shows the effects of SAA resource recovery Commit and Backout calls on CPI Communications conversation states for half-duplex conversations. Table C-5 on page 414 shows the effects of TX (Transaction Demarcation) calls on CPI Communications conversation states for full-duplex conversations. Table C-6 on page 416 shows the effects of SAA resource recovery calls on CPI Communications conversation states for full-duplex conversations.

C.1 How to Use the State Tables

Each CPI Communications call[4] 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 calls that are processed in non-blocking processing mode, the following special considerations apply:

- When a call gets the CM_OPERATION_INCOMPLETE return code, the operation remains in progress as an outstanding operation on the conversation (when conversation-level non-blocking is used) or on the queue with which the call is associated (when queue-level non-blocking is used). Any other calls (except Cancel_Conversation) on that conversation or queue 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.

The following special considerations apply for conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM:

- A state transition symbol ending with a caret (for example, 1^ or –^) means that the program may be in the **Backout-Required** condition following the call. (Note that the state change for the conversation is indicated by the *first* character of these symbols.)
- When a program is in the **Backout-Required** condition, its protected conversations are restricted from issuing certain CPI Communications calls. These calls are designated in the table with the symbol ↓'. Where this symbol appears, the call is valid in this state unless the conversation is protected and the program is in the **Backout-Required** condition. If the call is invalid, a *return_code* of CM_PROGRAM_STATE_CHECK is returned and no conversation state transition occurs.

4. Only the calls that affect conversation states are included in the State table.

C.1.1 Example

For an example of how the half-duplex 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_LEVEL 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 Section C.2.3 on page 386 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'' is never 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'') or CM_OPERATION_INCOMPLETE (''oi'') 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.

C.2 Explanation of Half-duplex 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.

C.2.1 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. • *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is set to CM_SYNC_POINT, but the conversation is not currently included in a transaction. 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. • *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* is set to CM_SYNC_POINT, but the conversation is not currently included in a transaction. 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 either *sync_level* is set to CM_NONE or the conversation is in **Initialize_Incoming** state. • *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, but the conversation is not currently included in a transaction. 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. • *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, but the conversation is not currently included in a transaction. For a Send_Data call, F means the following: • *send_type* is set to CM_SEND_AND_FLUSH.

Abbreviation	Meaning
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_LEVEL, *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and the conversation is currently included in a transaction. For a Prepare_To_Receive call, S means the following: • *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and the conversation is currently included in a transaction.
W	*receive_type* is set to CM_RECEIVE_AND_WAIT.

C.2.2 Conversation Queues ( )

The following abbreviations are used for conversation queues:

Abbreviation	Meaning
N	*conversation_queue* is set to CM_INITIALIZATION_QUEUE
Q	*conversation_queue* is set to one of the following: CM_SEND_RECEIVE_QUEUE CM_EXPEDITED_SEND_QUEUE CM_EXPEDITED_RECEIVE_QUEUE

C.2.3 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_SECURITY_NOT_VALID CM_SECURITY_NOT_SUPPORTED CM_SEND_RCV_MODE_NOT_SUPPORTED CM_CONVERSATION_TYPE_MISMATCH CM_PIP_NOT_SPECIFIED_CORRECTLY CM_RETRY_LIMIT_EXCEEDED CM_SYNC_LVL_NOT_SUPPORTED_PGM CM_SYNC_LVL_NOT_SUPPORTED_SYS CM_TPN_NOT_RECOGNIZED CM_TP_NOT_AVAILABLE_NO_RETRY CM_TP_NOT_AVAILABLE_RETRY CM_RESOURCE_FAILURE_NO_RETRY CM_RESOURCE_FAILURE_RETRY For any other call, ae means one of the following: CM_CONVERSATION_TYPE_MISMATCH CM_PIP_NOT_SPECIFIED_CORRECTLY CM_SECURITY_NOT_VALID CM_SEND_RCV_MODE_NOT_SUPPORTED CM_SYNC_LVL_NOT_SUPPORTED_PGM CM_SYNC_LVL_NOT_SUPPORTED_SYS 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 or CM_SYNC_POINT_NO_CONFIRM.
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 or CM_SYNC_POINT_NO_CONFIRM 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 or CM_SYNC_POINT_NO_CONFIRM.

Abbreviation	Meaning
ed	ed means one of the following: CM_EXP_DATA_NOT_SUPPORTED CM_BUFFER_TOO_SMALL CM_CONVERSATION_ENDING
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
ns	CM_NO_SECONDARY_INFORMATION
oi	CM_OPERATION_INCOMPLETE
ok	CM_OK
pb	CM_INCLUDE_PARTNER_REJECT_BO
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
pn	CM_PARM_VALUE_NOT_SUPPORTED
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
sc	CM_PROGRAM_STATE_CHECK
se	CM_SYSTEM_EVENT
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. If conversation-level non-blocking is being used on a conversation, a program receives CM_OPERATION_NOT_ACCEPTED when it issues any call (except Cancel_Conversation) on the conversation while a previous operation is still in progress, regardless of the state. If conversation-level non-blocking is not being used on a conversation, a program receives CM_OPERATION_NOT_ACCEPTED when it issues any call associated with a queue that has a previous operation still in progress, regardless of the state. No conversation state transition occurs.
3. The CM_CALL_NOT_SUPPORTED return code is not included in the state table. It is returned when the local system provides an entry point for the call but does not support the function requested by the call, regardless of the state. No state transition occurs.

C.2.4 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
*	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
jt	CM_JOIN_TRANSACTION
no	CM_NO_STATUS_RECEIVED
po	CM_PREPARE_OK
se	CM_SEND_RECEIVED
tc	CM_TAKE_COMMIT or CM_TAKE_COMMIT_DATA_OK. These values are returned only for conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.
td	CM_TAKE_COMMIT_DEALLOCATE or CM_TAKE_COMMIT_DEALLOC_DATA_OK. These values are returned only for conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.
ts	CM_TAKE_COMMIT_SEND or CM_TAKE_COMMIT_SEND_DATA_OK. These values are returned only for conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.

C.2.5 Table Symbols for the Half-duplex State Table

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 does not allow this input or never returns the indicated return codes for this input in this state.
–	Remain in current state
1-18	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 a conversation not using *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, or not currently included in a transaction, this is equivalent to ↓. If the conversation has *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and the conversation is currently included in a transaction, however, ↓' means it is valid to make this call from this state unless the program is in the **Backout-Required** *condition*. In that case, the call is invalid and CM_PROGRAM_STATE_CHECK is returned. For valid calls, see the table entries immediately below this symbol to determine the state transition resulting from the call.
^	For a conversation not using *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM or not currently included in a transaction, this symbol should be ignored. For a conversation using *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and currently included in a transaction, when this symbol follows a state number or a – (for example, 1^ or –^), it means the program may be in the **Backout-Required** condition following the call.
#	A conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and currently included in a transaction goes to the state it was 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_Completion and Wait_For_Conversation can only be issued when one or more calls have received a *return_code* of CM_OPERATION_INCOMPLETE. When Wait_For_Completion or Wait_For_Conversation completes with a *return_code* of CM_OK, it indicates one or more conversations on which an operation has completed. Each of those conversations then moves to the appropriate state as determined by the return code for the operation that is now completed and by the other factors that determine state transitions.
?	For programs using the TX (Transaction Demarcation) interface with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the state of the conversation with respect to the transaction is unknown.

C.3 Half-duplex State Table

Table C-1 States and Transitions for CPI Communications Calls: Half-duplex

	States 1–8 and 14 are used by all conversations								**Used only by conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM**						
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**	**Prepared 18**	**Initialize-Incoming 14**
Accept_Conversation	↓	/	/	/	/	/	/	/	/	/	/	/	/	/	/
[ok] [da,sc]	4 –														
Accept_Incoming	[pc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	↓
[ok] [da] [oi,pc,sc]															4 1 –
Allocate	[pc]	↓'	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok] [ae] [oi,pc,pe,un,sc]		3 1 –													
Cancel_Conversation	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok] [pc]		1^ –	1^ –	1^ –	1^ –	1^ –	1^ –	1^ –	1^ –	1^ –	1^ –	1^ –	1^ –	1^ –	1^ –
Confirm	[pc]	[sc]	↓'	[sc]	↓'	[sc]	[sc]	[sc]	↓'	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok] [ae] [da,rf] [bo] [db,pb,rb] [ep] [oi,pc] [sc]			– 1 1 –^ 1^ 4 – –		3 / 1 3^ 1^ 4 – /				4 1 1 4^ 1^ 4 – /						
Confirmed	[pc]	[sc]	[sc]	[sc]	[sc]	↓'	↓'	↓'	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok] [oi,pc]						4 –	3 –	1 –							
Deallocate(A)	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	/
[ok] [oi,pc]		1 –	1^ –	1^ –	1^ –	1^ –	1^ –	1^ –	1^ –	1^ –	1^ –	1^ –	1^ –	1^ –	
Deallocate(C)	[pc]	[sc]	↓	[sc]	↓	[sc]	[sc]	[sc]	/	/	/	/	/	/	/
[ok,da,rf] [ae] [ep] [oi,pc] [sc]			1 1 4 – –		1 / 4 – /										
Deallocate(F)	[pc]	[sc]	↓	[sc]	↓	[sc]	[sc]	[sc]	/	/	/	/	/	/	↓
[ok] [oi,pc] [sc]			1 – –		1 – /										1 – /
Deallocate(S)	[pc]	[sc]	↓'	[sc]	↓'	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	/
[ok] [oi,pc] [sc]			10 – –		10 – /										

	States 1–8 and 14 are used by all conversations								Used only by conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM						
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**	**Prepared 18**	**Initialize-Incoming 14**
Deferred_Deallocate†	[pc]	[sc]	↓'	[sc]	↓'	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]			–‡		–‡										
[ae]			1		/										
[ep]			4		4										
[oi,pc]			–		–										
[db,rb]			1^		1^										
[pb]			1^		/										
[bo]			–^		3^										
Extract_AE_Qualifier	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Extract_AP_Title	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Extract_Appl_Ctx_Name	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Extract_Conv_State	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	–
[bo]		/	–	–	–	–	–	/	–	–	–	–	–	–	/
Extract_Conv_Type	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Extract_Init_Data	[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]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Extract_Secondary_Info	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,ns,pc]	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Extr_Send_Rcv_Mode	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Extract_Sync_Level	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Extract_TP_Name	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Extract_Transaction_Control	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Flush	[pc]	[sc]	↓'	[sc]	↓'	[sc]	[sc]	[sc]	↓'	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]			–		3				4						
[oi,pc]			–		–				–						
Include_Ptr_In_Trans†	[pc]	[sc]	↓'	[sc]	↓'	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]			–		3										
[oi,pc,sc]			–		–										
[ae]			1		/										
[ep]			4		4										
[da,rf]			1		1										
Init_Conversation§	↓	/	/	/	/	/	/	/	/	/	/	/	/	/	/
[ok]	2														
[pc]	–														
Init_For_Incoming	↓	/	/	/	/	/	/	/	/	/	/	/	/	/	/
[ok]	14														

	States 1–8 and 14 are used by all conversations								Used only by conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM						
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**	**Prepared 18**	**Initialize-Incoming 14**
Prepare	[pc]	[sc]	↓’	[sc]	↓’	[sc]	[sc]	[sc]	↓’	↓’	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]			18		18				18	18					
[ae,rf]			1		/				1	1					
[ep]			4		4				4	4					
[oi,pc,sc]			–		–				–	–					
[db,rb,pb]			1^		1^				1^	1^					
[bo]			–^		–^				–^	–^					
Prepare_To_Receive(C)	[pc]	[sc]	↓’	[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,pb,rb]			1^		1^										
[oi,pc]			–		–										
[sc]			–		/										
Prepare_To_Receive(F)	[pc]	[sc]	↓’	[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]	[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,no}				–										–	
[ok] {nd,se}				3										/	
[ok] {dr,se}				5										/	
[ok] {*,co}				6										/	
[ok] {*,cs}				7										/	
[ok] {*,cd}				8										/	
[ok] {*,jt}				–										/	
[ok] {*,po}				/										–	
[ok] {*,tc}				11										/	
[ok] {*,ts}				12										/	
[ok] {*,td}				13										/	
[ae]				1										1	
[da,dn,rf]				1										/	
[bo]				–^										–^	
[db,pb,rb]				1^										1^	
[en,ep]				–										4	
[et]				–										/	
[pc,sc,un]				–										–	

	States 1–8 and 14 are used by all conversations								**Used only by conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM**						
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**	**Prepared 18**	**Initialize-Incoming 14**
Receive(W)	[pc]	[sc]	↓'	↓'	↓'	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	↓'	[sc]
[ok] {dr,no}			4	–	4									–	
[ok] {nd,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] {*,jt}			–	–	/									/	
[ok] {*,po}			/	/	/									–	
[ok] {*,tc}			11	11	11									/	
[ok] {*,ts}			12	12	12									/	
[ok] {*,td}			13	13	13									/	
[ae]			1	1	/									1	
[bo]			4^	–^	4^									–^	
[da,dn,rf]			1	1	1									/	
[db,pb,rb]			1^	1^	1^									1^	
[en,ep]			4	–	4									4	
[et]			/	–	/									/	
[oi,pc]			–	–	–									–	
[sc]			–	–	/									–	
Rcv_Exp_Data	[pc]	[sc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,ed,oi,pc]			–	–	–	–	–	–	–	–	–	–	–	–	
Request_To_Send+	[pc]	[sc]	↓'	↓'	↓'	↓'	↓'	↓'	[sc]	[sc]	↓'	↓'	↓'	↓'	[sc]
[oi,ok,pc]			–	–	–	–	–	–			–	–	–	–	
[sc]			/	–	/	/	/	/			/	/	/	/	
Send_Data	[pc]	[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^						1^	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^						1^	1^	1^		
[ep]			4		4						/	/	/		
[oi,pc]			–		–						–	–	–		
[pb]			1^		/						/	/	/		
[sc]			–		/						–	–	–		

	States 1–8 and 14 are used by all conversations								**Used only by conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM**						
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**	**Prepared 18**	**Initialize-Incoming 14**
Send_Error	[pc]	[sc]	↓’	↓’	↓’	↓’	↓’	↓’	[sc]	[sc]	↓’	↓’	↓’	[sc]	[sc]
[ok]			–	3	3	3	3	3			3	3	3		
[ae,da]			1	1	/	/	/	/			/	/	/		
[bo]			–^	–^	3^	/		/			–^	–^	–^		
[db]			1^	1^	/	/	/	/			/	/	/		
[dn]			/	1	/	/	/	/			/	/	/		
[dnb]			/	1^	/	/	/	/			/	/	/		
[ep]			4	–	/	/	/	/			/	/	/		
[oi,pc]			–	–	–	–	–	–			–	–	–		
[pb]			1^	1^	/	/	/	/			/	/	/		
[rb]			1^	1^	1^	1^	1^	1^			1^	1^	1^		
[rf]			1	1	1	1	1	1			1	1	1		
[sc]			/	–	/	/	/	/			/	/	/		
Send_Exp_Data	[pc]	[sc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,ed,oi,pc]			–	–	–	–	–	–	–	–	–	–	–	–	
Set_AE_Qualifier	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–													
Set_Allocate_Confirm	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–													
Set_AP_Title	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–													
Set_Appl_Context_Name	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–													
Set_Begin_Transaction	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Set_Confirmation_Urgency	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	–
Set_Conv_Sec_PW	[pc]	↓	[sc]	[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]	[sc]
[ok,pc,pn]		–													
Set_Conv_Sec_User_ID	[pc]	↓	[sc]	[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]	[sc]
[ok,pc]		–													
Set_Deallocate_Type	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Set_Error_Direction	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	–
Set_Fill	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	–
Set_Initialization_Data	[pc]	↓	[sc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–		–											
Set_Join_Transaction	[pc]	[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]	[sc]
[ok,pc]		–													
Set_Partner_LU_Name	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–													

	States 1–8 and 14 are used by all conversations								Used only by conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM						
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**	**Prepared 18**	**Initialize-Incoming 14**
Set_Prep_Data_Permitted	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Set_Prep_To_Rcv_Type	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Set_Processing_Mode	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	–
Set_Q_Callback_Func(N)	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	↓
[ok,pc]		–													–
Set_Q_Callback_Func(Q)	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Set_Q_Proc_Mode(N)	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	↓
[ok,pc]		–													–
Set_Q_Proc_Mode(Q)	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Set_Receive_Type	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	–
Set_Return_Control	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–													
Set_Send_Rcv_Mode	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–													
Set_Send_Type	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Set_Sync_Level	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc,pn]		–													
Set_TP_Name	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–													
Set_Transaction_Control	[pc]	↓	[sc]	[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]	[sc]
[ok,pc]			–	–	–				–	–					
Wait_For_Completion	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok]	/	%	%	%	%	%	%	%	%	%	%	%	%	%	%
[pc]	/	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Wait_For_Conversation	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok]	/	%	%	%	%	%	%	%	%	%	%	%	%	%	%
[sc,se]	/	–	–	–	–	–	–	–	–	–	–	–	–	–	–

Notes:

† The state table entries for Deferred_Deallocate and Include_Partner_In_Transaction calls are for conversations using an OSI TP CRM only. These calls get the CM_PROGRAM_PARAMETER_CHECK if issued on a conversation using an LU 6.2 CRM, regardless of the state.

‡ CPI Communications suspends action on the Deferred_Deallocate call until the transaction is committed or backed out.

§ 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 Section 3.7.1 on page 26.

+ The Request_To_Send call is not allowed in **Send, Send-Pending, Confirm-Send, Confirm-Deallocate, Sync-Point, Sync-Point-Send, Sync-Point-Deallocate** or **Prepared** state when the conversation is using an OSI TP CRM. The call gets the CM_PROGRAM_STATE_CHECK return code.

C.4 Effects of Calls on Half-duplex Conversations to X/Open TX Interface

Table C-2 shows the state transitions resulting from calls to the TX (Transaction Demarcation) interface on half-duplex conversations. This table applies only to conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.

The following abbreviations are used for return codes in Table C-2.

Abbreviation	Meaning
bo	[TX_ROLLBACK] or [TX_ROLLBACK_NO_BEGIN]
coh	[TX_COMMITTED] or [TX_COMMITTED_NO_BEGIN]
com	[TX_MIXED] or [TX_MIXED_NO_BEGIN]
cop	[TX_HAZARD] or [TX_HAZARD_NO_BEGIN]
fa	[TX_FAIL]
ok	[TX_OK] or [TX_NO_BEGIN]
sc	[TX_PROTOCOL_ERROR]

Table C-2 States and Transitions for Protected Half-duplex (X/Open TX)

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	Prepared 18	Initialize-Incoming 14
tx_commit( ) call	[sc]†	↓‡	↓	[sc]	↓	[sc]	[sc]	↓‡	↓	↓	↓	↓	↓	↓	↓‡
[ok,cop,com]		–	–◆		3			–	4	1	4	3	1	3§	–
[bo]		–	#		#			–	#	#	#	#	#	#	–
[sc]		–	–		–			–	–	–	–	–	–	–	–
[fa]		–	?		?			–	?	?	?	?	?	?	–
tx_rollback( ) call	↓+	↓‡	↓	↓	↓	↓	↓	↓‡	↓	↓	↓	↓	↓	↓	↓‡
[ok,com,cop,coh]	–	–	#	#	#	#	#	–	#	#	#	#	#	#	–
[sc]	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
[fa]		–	?	?	?	?	?	–	?	?	?	?	?	?	–
tx_begin( ) call	–	–	–	–	–	–	–	–	–	–	/	/	/	/	–
tx_close( ) call	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
tx_info( ) call	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
tx_open( ) call	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
tx_set_commit_return( ) call	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
tx_set_trans_control( ) call	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
tx_set_trans_timeout( ) call	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

Notes:

† When a program started by an incoming conversation startup request issues a *tx_commit*( ) call before issuing an Accept_Conversation call, a state check results. The *tx_commit*( ) call has no effect on other conversations in **Reset** state.

‡ Conversations in **Initialize**, **Confirm-Deallocate** or **Initialize-Incoming** state are not affected by *tx_commit*( ) and *tx_rollback*( ) calls.

§ The conversation goes to **Reset** state if the local program had issued a Deferred_Deallocate call or a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL prior to entering **Prepared** state. The

conversation goes to **Receive** state if the local program had issued a Prepare_To_Receive call with *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, respectively, prior to entering Prepared state.

+ When a program started by an incoming conversation startup request issues a *tx_rollback*( ) call before issuing an Accept_Conversation call, the underlying conversation is actually backed out, though the CPI Communications conversation remains in **Reset** state.

♦ The conversation goes to **Reset** state if the program had issued a Deferred_Deallocate call prior to issuing the *tx_commit*( ) call.

C.5 Effects of Calls to the SAA RRI on Half-duplex Conversations

Table C-3 shows the state transitions resulting from calls to the SAA resource recovery interface on half-duplex conversations. This table applies only to conversations with *sync_level* set to CM_SYNC_POINT.

The following abbreviations are used for return codes in Table C-3.

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

Table C-3 States and Transitions for Half-duplex Protected (CPIRR)

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]†	↓§	↓	[sc]	↓	[sc]	[sc]	↓§	↓	↓	↓	↓	↓	↓§
[ok,cop,com]		–	–+		3			–	4	1	4	3	1	–
[bo,bop,bom]		–	#		#			–	#	#	#	#	#	–
[sc]		–	–		–			–	–	–	–	–	–	–
Backout call	↓‡	↓§	↓	↓	↓	↓	↓	↓§	↓	↓	↓	↓	↓	↓§
[ok,bop,bom]	–	–	#	#	#	#	#	–	#	#	#	#	#	–

Notes:

† When a program started by an incoming conversation startup 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.

‡ When a program started by an incoming conversation startup request issues a Backout call before issuing an Accept_Conversation call, the underlying conversation is actually backed out, though the CPI Communications conversation remains in **Reset** state.

§ Conversations in **Initialize**, **Confirm-Deallocate** or **Initialize-Incoming** state are not affected by Commit and Backout calls.

\+ The conversation goes to **Reset** state if the program had issued a Deferred_Deallocate call prior to issuing the Commit call.

C.6 Explanation of Full-duplex 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.

C.6.1 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	*deallocate_type* is set to CM_DEALLOCATE_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 either *sync_level* is set to CM_NONE or the conversation is in **Initialize_Incoming** state. • *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, but the conversation is not currently included in a transaction. 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
S	For a Deallocate call, S means the following: • *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM and the conversation is currently included in a transaction.
W	*receive_type* is set to CM_RECEIVE_AND_WAIT.

C.6.2 Conversation Queues ( )

The following abbreviations are used for conversation queues:

Abbreviation	Meaning
N	*conversation_queue* is set to CM_INITIALIZATION_QUEUE
Q	*conversation_queue* is set to one of the following: CM_SEND_QUEUE CM_RECEIVE_QUEUE CM_EXPEDITED_SEND_QUEUE CM_EXPEDITED_RECEIVE_QUEUE

C.6.3 Return Code Values [ ]

The following table shows abbreviations that are used for return codes. The state table for CPI Communications calls on full-duplex conversations follows.

Abbreviation	Meaning
ae	For an Allocate call, ae means one of the following: CM_ALLOCATE_FAILURE_NO_RETRY CM_ALLOCATE_FAILURE_RETRY CM_SECURITY_NOT_VALID CM_SECURITY_NOT_SUPPORTED CM_SEND_RCV_MODE_NOT_SUPPORTED CM_CONVERSATION_TYPE_MISMATCH CM_PIP_NOT_SPECIFIED_CORRECTLY CM_RETRY_LIMIT_EXCEEDED CM_SYNC_LVL_NOT_SUPPORTED_PGM CM_SYNC_LVL_NOT_SUPPORTED_SYS CM_TPN_NOT_RECOGNIZED CM_TP_NOT_AVAILABLE_NO_RETRY CM_TP_NOT_AVAILABLE_RETRY CM_RESOURCE_FAILURE_NO_RETRY CM_RESOURCE_FAILURE_RETRY For any other call, ae means one of the following: CM_ALLOCATION_ERROR CM_CONVERSATION_TYPE_MISMATCH CM_SEND_RCV_MODE_NOT_SUPPORTED CM_PIP_NOT_SPECIFIED_CORRECTLY CM_SECURITY_NOT_VALID CM_SYNC_LVL_NOT_SUPPORTED_SYS 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_NO_CONFIRM.
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_NO_CONFIRM and means one of the following: CM_DEALLOCATED_ABEND_BO CM_DEALLOCATED_ABEND_SVC_BO CM_DEALLOCATED_ABEND_TIMER_BO
dn	CM_DEALLOCATED_NORMAL
dr	CM_DEALLOC_CONFIRM_REJECT

Abbreviation	Meaning
ds	CM_CONV_DEALLOC_AFTER_SYNCPT. This return code is returned only for conversations with *sync_level* set to CM_SYNC_POINT_NO_CONFIRM.
ed	This return code is reported for expedited-data calls only. ed means one of the following: CM_EXP_DATA_NOT_SUPPORTED CM_BUFFER_TOO_SMALL CM_CONVERSATION_ENDING
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
ns	CM_NO_SECONDARY_INFORMATION
oi	CM_OPERATION_INCOMPLETE
ok	CM_OK
pb	CM_INCLUDE_PARTNER_REJECT_BO
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
pn	CM_PARM_VALUE_NOT_SUPPORTED
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
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 a call associated with a queue that has a a previous operation still in progress, regardless of the state. No conversation state transition occurs.
3. The CM_CALL_NOT_SUPPORTED return code is not included in the state table. It is returned when the local system provides an entry point for the call but does not support the function requested by the call, regardless of the state. No state transition occurs.

C.6.4 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
*	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
jt	CM_JOIN_TRANSACTION
no	CM_NO_STATUS_RECEIVED
po	CM_PREPARE_OK
tc	CM_TAKE_COMMIT or CM_TAKE_COMMIT_DATA_OK. These values are returned only for conversations with *sync_level* set to CM_SYNC_POINT_NO_CONFIRM.
td	CM_TAKE_COMMIT_DEALLOCATE or CM_TAKE_COMMIT_DEALLOC_DATA_OK. These values are returned only for conversations with *sync_level* set to CM_SYNC_POINT_NO_CONFIRM.

C.6.5 Table Symbols for the Full-duplex State Table

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
1-18	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.
%	Wait_for_Completion (CMWCMP) can only be issued when one or more of the calls issued by the program has received a *return code* of CM_OPERATION_INCOMPLETE. When Wait_for_Completion completes with a return code of CM_OK, it returns a list of outstanding-operation-IDs that identify the operations that have completed. Each of the conversations on which a call has completed then makes a transition to the appropriate state as indicated for the operations that are now complete.
↓'	For a conversation with *sync_level* set to CM_NONE or not currently included in a transaction, this is equivalent to ↓. If the conversation has *sync_level* set to CM_SYNC_POINT_NO_CONFIRM and the conversation is currently included in a transaction, however, ↓' means it is valid to make this call from this state unless the program is in the **Backout-Required** condition. In that case, the call is invalid and CM_PROGRAM_STATE_CHECK is returned. For valid calls, see the table entries immediately below this symbol to determine the state transition resulting from the call.
^	For a conversation with *sync_level* set to CM_NONE or not currently included in a transaction, this symbol should be ignored. For a conversation using *sync_level* set to CM_SYNC_POINT_NO_CONFIRM and currently included in a transaction, when this symbol follows a state number or a – (for example, 1^ or –^), it means the program may be in the **Backout-Required** condition following the call.
#	Conversations with *sync_level* set to CM_SYNC_POINT_NO_CONFIRM go to the state they were in at the completion of the most recent synchronization point. If there was no prior synchronization event, both sides of the conversation go to **Send-Receive** state.

Note: The following calls can only be issued on half-duplex conversations. When issued on full-duplex conversations, CM_PROGRAM_PARAMETER_CHECK is returned for all conversation states except Reset. Therefore, Confirm, Set_Error_Direction, Prepare_To_Receive, Set_Prepare_To_Receive_Type, Request_To_Send, Set_Processing_Mode, Set_Confirmation_Urgency and Test_Request_To_Send_Received are not shown in the state tables.

C.7 Full-duplex State Table

Table C-4 States and Transitions for CPI Communications Calls: Full-duplex

	Used by CPI–C FDX conversations										
	Reset	**Ini-tialize**	**Ini-tialize-Incoming**	**Send-Receive**	**Send-Only**	**Receive-Only**	**Defer-Dealloc**	**Sync-Point**	**Sync-Pt-Dealloc**	**Prepared**	**Confirm-Dealloc**[†]
Inputs	**1**	**2**	**14**	**17**	**15**	**16**	**10**	**11**	**13**	**18**	**8**
Accept_Conversation	↓	/	/	/	/	/	/	/	/	/	/
[ok] [da,sc]	17 –										
Accept_Incoming	[pc]	[sc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok] [da] [oi,pc,sc]			17 1 –								
Allocate	[pc]	↓'	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok] [ae] [oi,pc,pe,un,sc]		17 1 –									
Cancel_Conversation	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok] [pc]		1^ –	1^ –	1^ –	1^ –	1^ –	1^ –	1^ –	1^ –	1^ –	1^ –
Confirmed[‡]	[pc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	↓
[ok] [oi,pc]											1 –
Deallocate(A)	[pc]	↓	/	↓	↓	↓	↓	↓	↓	↓	↓
[ok] [oi,pc]		1^ –		1^ –	1^ –	1^ –	1^ –	1^ –	1^ –	1^ –	1^ –
Deallocate(C)[‡]	[pc]	[sc]	/	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,ae,da,dn,rf] [dr,oi,pc,sc]				16 –							
Deallocate(F)	[pc]	[sc]	↓	↓	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok] [da,dn,rf] [ae] [oi,pc] [sc]			1 / / – /	16 16 16 – –	1 1 / –						
Deallocate(S)	[pc]	[sc]	/	↓'	/	/	[sc]	[sc]	[sc]	[sc]	[sc]
[ok] [ae,oi,pc,sc] [db,rb] [bo] [ds] [pb]				10 – 1^ –^ 1 1^							
Deferred_Deallocate[§]	[pc]	[sc]	[sc]	↓'	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok] [oi,pc] [ae] [db,rb,pb] [bo]				–[+] – – 1^ –^							
Extract_AE_Qualifier	[pc]	↓	[sc]	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–		–	–	–	–	–	–	–	–
Extract_AP_Title	[pc]	↓	[sc]	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–		–	–	–	–	–	–	–	–
Extract_Appl_Ctx_Name	[pc]	↓	[sc]	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–		–	–	–	–	–	–	–	–

	Used by CPI–C FDX conversations										
	Reset	**Ini-tialize**	**Ini-tialize-Incoming**	**Send-Receive**	**Send-Only**	**Receive-Only**	**Defer-Dealloc**	**Sync-Point**	**Sync-Pt-Dealloc**	**Prepared**	**Confirm-Dealloc**†
Inputs	**1**	**2**	**14**	**17**	**15**	**16**	**10**	**11**	**13**	**18**	**8**
Extract_Conv_State	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–
[bo]		/	/	–	/	/	–	–	–	–	/
Extract_Conv_Type	[pc]	↓	[sc]	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–		–	–	–	–	–	–	–	–
Extract_Init_Data	[pc]	[sc]	[sc]	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]				–	–	–	–	–	–	–	–
Extract_Mode_Name	[pc]	↓	[sc]	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–		–	–	–	–	–	–	–	–
Extract_Part_LU_Name	[pc]	↓	[sc]	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–		–	–	–	–	–	–	–	–
Extract_Sec_User_ID	[sc]	↓	[sc]	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–		–	–	–	–	–	–	–	–
Extract_Secondary_Info	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,ns,pc]	–	–	–	–	–	–	–	–	–	–	–
Extract_Send_Receive_Mode	[pc]	↓	[sc]	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–		–	–	–	–	–	–	–	–
Extract_Sync_Level	[pc]	↓	[sc]	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–		–	–	–	–	–	–	–	–
Extract_TP_Name	[pc]	↓	[sc]	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–		–	–	–	–	–	–	–	–
Extract_Transaction_Control	[pc]	↓	[sc]	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–		–	–	–	–	–	–	–	–
Flush	[pc]	[sc]	[sc]	↓’	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]				–	–						
[oi,pc]				–	–						
[rf,da,dn]				16	1						
[ae]				16	/						
[db,rb]				1^	/						
[bo]				–^	/						
[ds]				1	/						
[sc]				–	/						
[pb]				1^	/						
Include_Ptr_In_Trans§	[pc]	[sc]	[sc]	↓’	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]				–							
[oi,pc]				–							
[ae,da,rf,dn]				16							
Initialize_Conversation ♦	↓	/	/	/	/	/	/	/	/	/	/
[ok]	2										
[pc]	–										
Initialize_for_Incoming	↓	/	/	/	/	/	/	/	/	/	/
[ok]	17										
[pc]	–										
Prepare	[pc]	[sc]	[sc]	↓’	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]				18							
[oi,pc]				–							
[ae,rf]				–							
[db,pb,rb]				1^							
[ds]				1							
[bo]				–^							

	Used by CPI–C FDX conversations									
	Reset	**Ini-tialize**	**Ini-tialize-Incoming**	**Send-Receive**	**Send-Only**	**Receive-Only**	**Defer-Dealloc**	**Sync-Point**	**Sync-Pt-Dealloc**	**Prepared**	**Confirm-Dealloc†**
Inputs	**1**	**2**	**14**	**17**	**15**	**16**	**10**	**11**	**13**	**18**	**8**
Receive(I)	[pc]	[sc]	[sc]	↓’	[sc]	↓	[sc]	[sc]	[sc]	↓’	[sc]
[ok] {dr,no}				–		–				–	
[ae]				1		1				1	
[da,rf]				1		1				/	
[dn]				15		1				/	
[en,et]				–		–				17	
[ep]				–		–				/	
[pc,un]				–		–				–	
[db,pb,rb]				1^		/				1^	
[bo]				–^		/				–^	
[ok] {*,tc}				11		/				/	
[ok] {*,td}				13		/				/	
[ok] {*,cd}				8		/				/	
[ds]				1		/				/	
[ok] {*,po}				/		/				–	
[ok] {*,jt}				–		/				/	
[sc]				–		/				–	
Receive(W)	[pc]	[sc]	[sc]	↓’	[sc]	↓	[sc]	[sc]	[sc]	↓’	[sc]
[ok] {dr,no}				–		–				–	
[ae]				1		1				1	
[da,rf]				1		1				/	
[dn]				15		1				/	
[en,et]				–		–				17	
[ep]				–		–				/	
[oi,pc]				–		–				–	
[db,pb,rb]				1^		/				1^	
[bo]				–^		/				–^	
[ok] {*,tc}				11		/				/	
[ok] {*,td}				13		/				/	
[ok] {*,cd}				8		/				/	
[ds]				1		/				/	
[ok] {*,po}				/		/				–	
[ok] {*,jt}				–		/				/	
[sc]				–		/				–	
Receive_Exp_Data	[pc]	[sc]	[sc]	↓	↓	↓	↓	↓	↓	↓	↓
[ok,ed,oi,pc,un]				–	–	–	–	–	–	–	–
Send_Data	[pc]	[sc]	[sc]	↓’	↓	[sc]	[sc]	↓’	↓’	[sc]	[sc]
(B) [ok]				–	–			–	–		
(F) [ok]				–	–			–	–		
(D(A)) [ok]				1^	1			1^	1^		
(D(C)) [ok]				16	/			/	/		
(D(F)) [ok]				16	1			/	/		
(D(S)) [ok]				10	/			/	/		
[da,rf,dn]				16	1			/	/		
[ae]				16	/			/	/		
[dr,oi,pc]				–	–			–	–		
[db,rb]				1^	/			1^	1^		
[bo]				–^	/			–^	–^		
[ds]				1	/			/	/		
[sc]				–	/			–	–		
[pb]				1^	/			/	/		

	Used by CPI–C FDX conversations										
	Reset	**Ini-tialize**	**Ini-tialize-Incoming**	**Send-Receive**	**Send-Only**	**Receive-Only**	**Defer-Dealloc**	**Sync-Point**	**Sync-Pt-Dealloc**	**Prepared**	**Confirm-Dealloc**†
Inputs	**1**	**2**	**14**	**17**	**15**	**16**	**10**	**11**	**13**	**18**	**8**
Send_Error	[pc]	[sc]	[sc]	↓’	↓	[sc]	[sc]	[sc]	[sc]	[sc]	↓
[ok]				–	–						17
[da,rf,dn]				16	1						1
[ae]				16	/						/
[pc,oi]				–	–						–
[ds]				1	/						/
[sc]				–	/						/
[db,rb]				1^	/						/
[bo]				–^	/						/
[pb]				1^	/						/
Send_Expedited_Data	[pc]	[sc]	[sc]	↓	↓	↓	↓	↓	↓	↓	↓
[ok,ed,oi,pc]				–	–	–	–	–	–	–	–
Set_AE_Qualifier	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–									
Set_Allocate_Confirm	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–									
Set_AP_Title	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–									
Set_Appl_Context_Name	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–									
Set_Begin_Transaction	[pc]	↓	[sc]	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–		–	–	–	–	–	–	–	–
Set_Conv_Sec_PW	[pc]	↓	[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]
[ok,pc,pn]		–									
Set_Conv_Sec_User_ID	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–									
Set_Conversation_Type	[pc]	↓	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–	–								
Set_Deallocate_Type	[pc]	↓	[sc]	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–		–	–	–	–	–	–	–	–
Set_Fill	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–
Set_Initialization_Data	[pc]	↓	[sc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–		–							
Set_Log_Data	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–
Set_Mode_Name	[pc]	↓	[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]
[ok,pc]		–									
Set_Prep_Data_Permitted	[pc]	↓	[sc]	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–
Set_Processing_Mode	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[pc]		–	–	–	–	–	–	–	–	–	–
Set_Q_Callback_Func(N)	[pc]	↓	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–	–								
Set_Q_Callback_Func(Q)	[pc]	↓	[sc]	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–		–	–	–	–	–	–	–	–
Set_Q_Proc_Mode(N)	[pc]	↓	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–	–								

	Used by CPI–C FDX conversations										
	Reset	**Ini-tialize**	**Ini-tialize-Incoming**	**Send-Receive**	**Send-Only**	**Receive-Only**	**Defer-Dealloc**	**Sync-Point**	**Sync-Pt-Dealloc**	**Prepared**	**Confirm-Dealloc†**
Inputs	**1**	**2**	**14**	**17**	**15**	**16**	**10**	**11**	**13**	**18**	**8**
Set_Q_Proc_Mode(Q)	[pc]	↓	[sc]	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–		–	–	–	–	–	–	–	–
Set_Receive_Type	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–
Set_Return_Control	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–									
Set_Send_Receive_Mode	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–									
Set_Send_Type	[pc]	↓	[sc]	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–		–	–	–	–	–	–	–	–
Set_Sync_Level	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc,pn]		–									
Set_TP_Name	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–									
Set_Transaction_Control	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–									
Wait_For_Completion	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok]	/	%	%	%	%	%	%	%	%	%	%
[pc]	/	–	–	–	–	–	–	–	–	–	–

Notes:

† This state is entered only if the conversation is allocated using an OSI TP CRM.

‡ This call can be issued only if the conversation is allocated using an OSI TP CRM.

§ The state table entries for Deferred_Deallocate and Include_Partner_In_Transaction calls are for conversations using an OSI TP CRM only. These calls get the CM_PROGRAM_PARAMETER_CHECK if issued on a conversation using an LU 6.2 CRM, regardless of the state.

\+ CPI Communications suspends action on the Deferred_Deallocate call until the transaction is committed or backed out.

♦ 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 Section 3.7.1 on page 26.

C.8 Effects of Calls on Full-duplex Conversations to X/Open TX Interface

Table C-5 shows the state transitions resulting from calls to the TX (Transaction Demarcation) interface on full-duplex conversations. This table applies only to conversations with *sync_level* set to CM_SYNC_POINT_NO_CONFIRM.

The following abbreviations are used for return codes in Table C-5:

Abbreviation	Meaning
bo	[TX_ROLLBACK] or [TX_ROLLBACK_NO_BEGIN]
coh	[TX_COMMITTED] or [TX_COMMITTED_NO_BEGIN]
com	[TX_MIXED] or [TX_MIXED_NO_BEGIN]
cop	[TX_HAZARD] or [TX_HAZARD_NO_BEGIN]
fa	[TX_FAIL]
ok	[TX_OK] or [TX_NO_BEGIN]
sc	[TX_PROTOCOL_ERROR]

Table C-5 States and Transitions for Protected Full-duplex (X/Open TX)

	Used by CPI–C FDX conversations										
	Reset	**Ini-tialize**	**Ini-tialize-Incoming**	**Send-Receive**	**Send-Only**	**Receive-Only**	**Defer-Dealloc**	**Sync-Point**	**Sync-Pt-Dealloc**	**Prepared**	**Confirm-Dealloc**
Inputs	**1**	**2**	**14**	**17**	**15**	**16**	**10**	**11**	**13**	**18**	**8**
tx_commit( ) call	[sc]†	↓‡	↓‡	↓	[sc]	[sc]	↓	↓	↓	↓	[sc]
[ok,cop,com]		–	–	–♦			1	17	1	17§	
[bo]		–	–	–			17	17	17	17	
[sc]		–	–	–			–	–	–	–	
[fa]		–	–	–			–	–	–	–	
tx_rollback( ) call	↓+	↓‡	↓‡	↓	[sc]	[sc]	↓	↓	↓	↓	[sc]
[ok,com,cop,coh]	–	–	–	–			17	17	17	17	
[sc]	–	–	–	–			–	–	–	–	
[fa]		–	–	–			–	–	–	–	
tx_begin( ) call	–	–	–	–	–	–	–	–	/	/	/
tx_close( ) call	–	–	–	–	–	–	–	–	–	–	–
tx_info( ) call	–	–	–	–	–	–	–	–	–	–	–
tx_open( ) call	–	–	–	–	–	–	–	–	–	–	–
tx_set_commit_return( ) call	–	–	–	–	–	–	–	–	–	–	–
tx_set_trans_control( ) call	–	–	–	–	–	–	–	–	–	–	–
tx_set_trans_timeout( ) call	–	–	–	–	–	–	–	–	–	–	–

Notes:

† When a program started by an incoming conversation startup request issues a *tx_commit*( ) call before issuing an Accept_Conversation call, a state check results. The *tx_commit*( ) call has no effect on other conversations in **Reset** state.

‡ Conversations in **Initialize** or **Initialize-Incoming** state are not affected by *tx_commit*( ) and *tx_rollback*( ) calls.

§ The conversation goes to **Reset** state if the local program had issued a Deferred_Deallocate call or a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL prior to entering **Prepared** state.

\+ When a program started by an incoming conversation startup request issues a *tx_rollback*( ) call before issuing an Accept_Conversation call, the underlying conversation is actually backed out, though the CPI Communications conversation remains in **Reset** state.

♦ The conversation goes to **Reset** state if the program had issued a Deferred_Deallocate call prior to issuing the *tx_commit*( ) call.

C.9 Effects of Calls to the SAA RRI on Full-duplex Conversations

Table C-6 shows the state transitions resulting from calls to the SAA resource recovery interface on full-duplex conversations. This table applies only to conversations with *sync_level* set to CM_SYNC_POINT_NO_CONFIRM.

Commit and Backout are resource recovery calls. Their return codes are as follows.

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

Table C-6 States and Transitions for Protected Full-duplex (CPIRR)

	Used by CPI–C FDX conversations										
	Reset	**Ini-tialize**	**Ini-tialize-Incoming**	**Send-Receive**	**Send-Only**	**Receive-Only**	**Defer-Dealloc**	**Sync-Point**	**Sync-Pt-Dealloc**	**Prepared**	**Confirm-Dealloc**
Inputs	**1**	**2**	**14**	**17**	**15**	**16**	**10**	**11**	**13**	**18**	**8**
Commit call†	[sc]	↓§	↓§	↓	[sc]	[sc]	↓	↓	↓	↓	[sc]
[ok,cop,com]		–	–	–+			1	17	1	17	
[bo,bop,bom]		–	–	–			17	17	17	17	
[fa,sc]		–	–	–			–	–	–	–	
Backout call‡	↓	↓§	↓§	↓	[sc]	[sc]	↓	↓	↓	↓	[sc]
[bo,bop,bom]	–	–	–	–			17	17	17	17	

Notes:

† When a program started by an incoming conversation startup 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.

‡ When a program started by an incoming conversation startup request issues a Backout call before issuing an Accept_Conversation call, the underlying conversation is actually backed out, though the CPI Communications conversation remains in **Reset** state.

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

\+ The conversation goes to **Reset** state if the program had issued a Deferred_Deallocate call prior to issuing the Commit call.

Appendix D

Mapping to OSI TP and LU 6.2 CRMs

This appendix is intended to help programmers match CPI Communications functions to equivalent OSI TP and LU 6.2 functions. It is divided into three sections, two for OSI TP (half-duplex and full-duplex respectively) and one for LU 6.2.

Some CPI Communications functions are not part of the OSI TP or LU 6.2 services. Specifically, these functions are provided by calls that map to local function only. The following is a list of such calls:

Convert_Incoming
Convert_Outgoing
Extract_Initialization_Data
Extract_Maximum_Buffer_Size
Extract_Secondary_Information
Release_Local_TP_Name
Set_Processing_Mode
Set_Queue_Callback_Function
Set_Queue_Processing_Mode
Specify_Local_TP_Name
Wait_For_Completion
Wait_For_Conversation.

Calls that are not specifically a part of OSI TP or LU 6.2 services are not listed in the CPI Communications to OSI TP and LU 6.2,respectively.

D.1 OSI TP CRMs (Half-duplex)

This section summarizes the CPI-C application service element (ASE) services, maps the services both to and from the OSI TP services, and defines the sequencing rules and state table for CPI-C use by OSI TP programs using half-duplex conversations.

The CPI Communications calls have been mapped to the OSI TP services described in ISO/IEC 10026-2, Information Technology — Open Systems Interconnection — Distributed Transaction Processing — Part 2: OSI TP Service Definition.

Programs written using CPI Communications can communicate with OSI TP programs. Two sets of tables beginning with Table D-1 on page 419 and Table D-16 on page 443 show the functional relationships between OSI TP services and CPI Communications calls. Use these tables to determine how the function of a particular OSI TP service is provided through CPI Communications for half-duplex conversations.

This section is intended for programmers who are familiar with OSI TP.

D.1.1 Summary of CPI-C ASE Services

The CPI-C ASE services are defined in Chapter 5.

D.1.2 Mapping CPI-C to OSI TP Services

The following tables present the complete mapping from the CPI-C calls to the OSI TP services. Some of the CPI-C calls provide local services only (such as CMSST, Set_Send_Type), and are not included in these tables. However, the effect of these calls can be determined from this mapping.

The following conventions are used within the tables in this section:

* The parameter is not directly supported by CPI-C.

=xxx The value xxx is always used.

n/a The value is not applicable in this case.

Table D-1 Mapping CPI-C Calls to OSI TP Services

CPI-C Call	OSI TP Service
CMALLC - Allocate (See Table D-2 on page 421 for details.)	TP-BEGIN-DIALOGUE request
CMCANC - Cancel_Conversation (See Table D-3 on page 424 for details.)	TP-BEGIN-DIALOGUE response TP-U-ABORT request
CMCFM - Confirm (See Table D-4 on page 425 for details.)	TP-BEGIN-TRANSACTION request TP-HANDSHAKE request TP-HANDSHAKE-AND-GRANT-CONTROL request
CMCFMD - Confirmed (See Table D-5 on page 426 for details.)	TP-HANDSHAKE response TP-HANDSHAKE-AND-GRANT-CONTROL response TP-END-DIALOGUE response
CMDEAL - Deallocate (See Table D-6 on page 427 for details.)	TP-BEGIN-DIALOGUE response TP-END-DIALOGUE request TP-DEFERRED-END-DIALOGUE request TP-U-ABORT request
CMDFDE - Deferred_Deallocate (See Table D-7 on page 429 for details.)	TP-DEFERRED-END-DIALOGUE request
CMFLUS - Flush Mapping (See Table D-8 on page 430 for details.)	TP-GRANT-CONTROL request
CMINCL - Include_Partner_In_Transaction (See Table D-9 on page 431 for details.)	TP-BEGIN-TRANSACTION request
CMPREP - Prepare (See Table D-10 on page 432 for details.)	TP-BEGIN-TRANSACTION request TP-PREPARE request
CMPTR - Prepare_to_Receive (See Table D-11 on page 433 for details.)	TP-BEGIN-TRANSACTION request TP-GRANT-CONTROL request TP-HANDSHAKE-AND-GRANT-CONTROL request TP-DEFERRED-GRANT-CONTROL request
CMRTS - Request_to_Send (See Table D-12 on page 435 for details.)	TP-REQUEST-CONTROL request
CMRCV - Receive (See Table D-13 on page 436 for details.)	TP-BEGIN-DIALOGUE response TP-BEGIN-TRANSACTION request TP-GRANT-CONTROL request
CMSEND - Send_Data (See Table D-14 on page 437 for details.)	TP-BEGIN-TRANSACTION request (TP-DATA request) UD-TRANSFER request TP-HANDSHAKE request TP-GRANT-CONTROL request TP-HANDSHAKE-AND-GRANT-CONTROL request TP-DEFERRED-GRANT-CONTROL request TP-END-DIALOGUE request TP-DEFERRED-END-DIALOGUE request TP-U-ABORT request
CMSERR - Send_Error (See Table D-15 on page 441 for details.)	TP-BEGIN-DIALOGUE response TP-BEGIN-TRANSACTION request TP-U-ERROR request

The following OSI TP services are not directly mapped to the CPI-C calls:

TP-COMMIT request
: Supported through a resource recovery interface.

TP-DONE request
: Not externalized to the application program.

TP-ROLLBACK request
: Supported through a resource recovery interface.

Table D-2 CMALLC — Allocate Mapping

CPI-C Call Parameters and Conversation Characteristics	OSI TP Service Parameters
if: *sync_level* = CM_NONE	
CMALLC - Allocate	TP-BEGIN-DIALOGUE request
*	Initiating-TPSU-Title
AP_title (characteristic)	Recipient-AP-Title
*	Recipient-API-Identifier
AE_qualifier (characteristic)	Recipient-AE-Qualifier
*	Recipient-AEI-Identifier
TP_name (characteristic)	Recipient-TPSU-Title
=Dialogue and Polarized	Functional-Units
mode_name (characteristic)	Quality-of-Service
application_context_name (characteristic)	Application-Context-Name
n/a	Begin-Transaction
allocate_confirm (characteristic)	Confirmation
initialization_data (characteristic)	User-Data
if: *sync_level* = CM_CONFIRM	
CMALLC - Allocate	TP-BEGIN-DIALOGUE request
*	Initiating-TPSU-Title
AP_title (characteristic)	Recipient-AP-Title
*	Recipient-API-Identifier
AE_qualifier (characteristic)	Recipient-AE-Qualifier
*	Recipient-AEI-Identifier
TP_name (characteristic)	Recipient-TPSU-Title
=Dialogue, Polarized, and Handshake	Functional-Units
mode_name (characteristic)	Quality-of-Service
application_context_name (characteristic)	Application-Context-Name
n/a	Begin-Transaction
allocate_confirm (characteristic)	Confirmation
initialization_data (characteristic)	User-Data
if: *sync_level* = CM_SYNC_POINT **and** *transaction_control* = CM_CHAINED_TRANSACTIONS	
CMALLC - Allocate	TP-BEGIN-DIALOGUE request
*	Initiating-TPSU-Title
AP_title (characteristic)	Recipient-AP-Title
*	Recipient-API-Identifier
AE_qualifier (characteristic)	Recipient-AE-Qualifier
*	Recipient-AEI-Identifier
TP_name (characteristic)	Recipient-TPSU-Title
=Dialogue, Polarized, Handshake, Commit, and Chained Transactions	Functional-Units
mode_name (characteristic)	Quality-of-Service
application_context_name (characteristic)	Application-Context-Name
n/a	Begin-Transaction
allocate_confirm (characteristic)	Confirmation
initialization_data (characteristic)	User-Data

CPI-C Call Parameters and Conversation Characteristics	**OSI TP Service Parameters**
if: *sync_level* = CM_SYNC_POINT **and** *transaction_control* = CM_UNCHAINED_TRANSACTIONS	
CMALLC - Allocate	TP-BEGIN-DIALOGUE request
*	Initiating-TPSU-Title
AP_title (characteristic)	Recipient-AP-Title
*	Recipient-API-Identifier
AE_qualifier (characteristic)	Recipient-AE-Qualifier
*	Recipient-AEI-Identifier
TP_name (characteristic)	Recipient-TPSU-Title
=Dialogue, Polarized, Handshake, Commit, and Unchained Transactions	Functional-Units
mode_name (characteristic)	Quality-of-Service
application_context_name (characteristic)	Application-Context-Name
begin_transaction (characteristic)	Begin-Transaction (See note at end of table.)
allocate_confirm (characteristic)	Confirmation
initialization_data (characteristic)	User-Data
if: *sync_level* = CM_SYNC_POINT_NO_CONFIRM **and** *transaction_control* = CM_CHAINED_TRANSACTIONS	
CMALLC - Allocate	TP-BEGIN-DIALOGUE request
*	Initiating-TPSU-Title
AP_title (characteristic)	Recipient-AP-Title
*	Recipient-API-Identifier
AE_qualifier (characteristic)	Recipient-AE-Qualifier
*	Recipient-AEI-Identifier
TP_name (characteristic)	Recipient-TPSU-Title
=Dialogue, Polarized, Commit, and Chained Transactions	Functional-Units
mode_name (characteristic)	Quality-of-Service
application_context_name (characteristic)	Application-Context-Name
n/a	Begin-Transaction
allocate_confirm (characteristic)	Confirmation
initialization_data (characteristic)	User-Data

CPI-C Call Parameters and Conversation Characteristics	**OSI TP Service Parameters**
if: *sync_level* = CM_SYNC_POINT_NO_CONFIRM **and** *transaction_control* = CM_UNCHAINED_TRANSACTIONS	
CMALLC - Allocate	TP-BEGIN-DIALOGUE request
*	Initiating-TPSU-Title
AP_title (characteristic)	Recipient-AP-Title
*	Recipient-API-Identifier
AE_qualifier (characteristic)	Recipient-AE-Qualifier
*	Recipient-AEI-Identifier
TP_name (characteristic)	Recipient-TPSU-Title
=Dialogue, Polarized, Commit, and Unchained Transactions	Functional-Units
mode_name (characteristic)	Quality-of-Service
application_context_name (characteristic)	Application-Context-Name
begin_transaction (characteristic)	Begin-Transaction (See note at end of table.)
allocate_confirm (characteristic)	Confirmation
initialization_data (characteristic)	User-Data
Note: if: begin_transaction = CM_BEGIN_IMPLICIT **and** *transaction_control* = CM_UNCHAINED_TRANSACTIONS **and** the program is in transaction mode **then** Begin-Transaction is set to true; **otherwise:** Begin-Transaction is set to false.	

Table D-3 CMCANC — Cancel_Conversation Mapping

CPI-C Call Parameters and Conversation Characteristics	**OSI TP Service Parameters**
if: Confirmation = true on TP-BEGIN-DIALOGUE indication **and** it is the first activity on the conversation following Accept_Conversation or Accept_Incoming	
CMCANC - Cancel_Conversation = rejected(user) *initialization_data* (characteristic)	TP-BEGIN-DIALOGUE response Result User-Data
otherwise	
CMCANC - Cancel_Conversation =null	TP-U-ABORT request User-Data

Table D-4 CMCFM — Confirm Mapping

CPI-C Call Parameters and Conversation Characteristics	OSI TP Service Parameters
if: *sync_level* = CM_SYNC_POINT **and** *transaction_control* = CM_UNCHAINED_TRANSACTIONS **and** *begin_transaction* = CM_BEGIN_IMPLICIT **and** it is the first activity on the conversation following the start of the current transaction	
CMCFM - Confirm	TP-BEGIN-TRANSACTION request (no parameters) TP-HANDSHAKE request
confirmation_urgency (characteristic)	Confirmation-Urgency
if: *sync_level* = CM_SYNC_POINT **and** the previous operation on the conversation was a Prepare_to_Receive with *prepare_to_receive_type* = CM_PREP_TO_RECEIVE_SYNC_LEVEL	
CMCFM - Confirm	TP-HANDSHAKE-AND-GRANT-CONTROL request (no parameters)
if: *sync_level* = CM_CONFIRM or *sync_level* = CM_SYNC_POINT	
CMCFM - Confirm	TP-HANDSHAKE request
confirmation_urgency (characteristic)	Confirmation-Urgency

Table D-5 CMCFMD — Confirmed Mapping

CPI-C Call Parameters and Conversation Characteristics	**OSI TP Service Parameters**
if in response to: *status_received* = CM_CONFIRM_RECEIVED	
CMCFMD - Confirmed	TP-HANDSHAKE response (no parameters)
if in response to: *status_received* = CM_CONFIRM_SEND_RECEIVED	
CMCFMD - Confirmed	TP-HANDSHAKE_AND-GRANT-CONTROL response (no parameters)
if in response to: *status_received* = CM_CONFIRM_DEALLOC_RECEIVED	
CMCFMD - Confirmed	TP-END-DIALOGUE response (no parameters)

Table D-6 CMDEAL — Deallocate Mapping

CPI-C Call Parameters and Conversation Characteristics	OSI TP Service Parameters
if: *deallocate_type* = CM_DEALLOCATE_SYNC_LEVEL **and** *sync_level* = CM_NONE	
CMDEAL - Deallocate =false	(See note 2 at end of table.) TP-END-DIALOGUE request Confirmation
if: *deallocate_type* = CM_DEALLOCATE_SYNC_LEVEL **and** *sync_level* = CM_CONFIRM	
CMDEAL - Deallocate =true	(See note 1 at end of table.) TP-END-DIALOGUE request Confirmation
if: *deallocate_type* = CM_DEALLOCATE_SYNC_LEVEL **and** (*sync_level* = CM_SYNC_POINT or *sync_level* = CM_SYNC_POINT_NO_CONFIRM) **and** (*transaction_control* = CM_CHAINED_TRANSACTIONS or (*transaction_control* = CM_UNCHAINED_TRANSACTIONS and the conversation is included in the current transaction))	
CMDEAL - Deallocate	(See note 3 at end of table.) TP-DEFERRED-END-DIALOGUE request (no parameters)
if: *deallocate_type* = CM_DEALLOCATE_SYNC_LEVEL **and** *sync_level* = CM_SYNC_POINT **and** (*transaction_control* = CM_UNCHAINED_TRANSACTIONS and the conversation is not currently included in the transaction)	
CMDEAL - Deallocate =true	(See note 1 at end of table.) TP-END-DIALOGUE request Confirmation
if: *deallocate_type* = CM_DEALLOCATE_SYNC_LEVEL **and** *sync_level* = CM_SYNC_POINT_NO_CONFIRM **and** (*transaction_control* = CM_UNCHAINED_TRANSACTIONS and the conversation is not currently included in the transaction)	
CMDEAL - Deallocate =false	(See note 2 at end of table.) TP-END-DIALOGUE request Confirmation
if: *deallocate_type* = CM_DEALLOCATE_FLUSH	
CMDEAL - Deallocate =false	(See note 2 at end of table.) TP-END-DIALOGUE request Confirmation
if: *deallocate_type* = CM_DEALLOCATE_CONFIRM	
CMDEAL - Deallocate =true	(See note 1 at end of table.) TP-END-DIALOGUE request Confirmation
if: *deallocate_type* = CM_DEALLOCATE_ABEND **and** it is the first activity on the conversation following Accept_Conversation or Accept_Incoming	
CMDEAL - Deallocate =rejected(user) *initialization_data* (characteristic)	TP-BEGIN-DIALOGUE response Result User-Data

CPI-C Call Parameters and Conversation Characteristics	OSI TP Service Parameters
if: *deallocate_type* = CM_DEALLOCATE_ABEND **and** it is not the first activity on the conversation following Accept_Conversation or Accept_Incoming	
CMDEAL - Deallocate *log_data* (characteristic)	TP-U-ABORT request User-Data
Note 1: if: Confirmation = true on TP-BEGIN-DIALOGUE indication **and** it is the first activity on the conversation following Accept_Conversation or Accept_Incoming **then** the identified OSI TP Service primitive will be preceded by TP-BEGIN-DIALOGUE response (Result = accepted and User-Data = *initialization_data* (characteristic)).	
Note 2: if: a TP-U-ERROR indication has arrived but has not been indicated to the program **then:** replace the TP-END-DIALOGUE request by a TP-U-ABORT request.	
Note 3: if: a TP-U-ERROR indication has arrived but has not been indicated to the program **then:** do not send any OSI TP Service primitive, a subsequent *tx_commit*() will generate a TP-ROLLBACK request.	

Table D-7 CMDFDE — Deferred_Deallocate Mapping

CPI-C Call Parameters and Conversation Characteristics	OSI TP Service Parameters
CMDFDE - Deferred_Deallocate	TP-DEFERRED-END-DIALOGUE request (no parameters)

Table D-8 CMFLUS — Flush Mapping

CPI-C Call Parameters and Conversation Characteristics	OSI TP Service Parameters
if: the previous operation on the conversation was a Prepare_to_Receive with *prepare_to_receive_type* = CM_PREP_TO_RECEIVE_SYNC_LEVEL	
CMFLUS - Flush	TP-GRANT-CONTROL request (no parameters)

Table D-9 CMINCL — Include_Partner_In_Transaction Mapping

CPI-C Call Parameters and Conversation Characteristics	OSI TP Service Parameters
CMINCL - Include_Partner_In_Transaction	TP-BEGIN-TRANSACTION request (no parameters)

Table D-10 CMPREP — Prepare Mapping

CPI-C Call Parameters and Conversation Characteristics	**OSI TP Service Parameters**
if: *sync_level* = CM_SYNC_POINT or *sync_level* = CM_SYNC_POINT_NO_CONFIRM **and** *transaction_control* = CM_UNCHAINED_TRANSACTIONS **and** *begin_transaction* = CM_BEGIN_IMPLICIT **and** it is the first activity on the conversation following the start of the current transaction	
CMPREP - Prepare	TP-BEGIN-TRANSACTION request (no parameters) TP-PREPARE request
prepare_data_permitted (characteristic)	Data-Permitted
if: *sync_level* = CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, **and** the previous operation on the conversation was a Prepare_To_Receive with *prepare_to_receive_type* = CM_PREP_TO_RECEIVE_SYNC_LEVEL	
CMPREP - Prepare	TP-DEFERRED-GRANT-CONTROL request (no parameters) TP-PREPARE request
prepare_data_permitted (characteristic)	Data-Permitted
if: *sync_level* = CM_SYNC_POINT or *sync_level* = CM_SYNC_POINT_NO_CONFIRM	
CMPREP - Prepare	TP-PREPARE request
prepare_data_permitted (characteristic)	Data-Permitted

Table D-11 CMPTR — Prepare_To_Receive Mapping

CPI-C Call Parameters and Conversation Characteristics	OSI TP Service Parameters
if: *prepare_to_receive_type* = CM_PREP_TO_RECEIVE_SYNC_LEVEL **and** *sync_level* = CM_NONE	
CMPTR - Prepare_To_Receive	(see note 3 at end of table.) TP-GRANT-CONTROL request (no parameters)
if: *prepare_to_receive_type* = CM_PREP_TO_RECEIVE_SYNC_LEVEL **and** *sync_level* = CM_CONFIRM	
CMPTR - Prepare_To_Receive confirmation_urgency (characteristic)	TP-HANDSHAKE-AND-GRANT-CONTROL request Confirmation-Urgency
if: *prepare_to_receive_type* = CM_PREP_TO_RECEIVE_SYNC_LEVEL **and** *sync_level* = CM_SYNC_POINT or *sync_level* = CM_SYNC_POINT_NO_CONFIRM **and** (*transaction_control* = CM_CHAINED_TRANSACTIONS or (*transaction_control* = CM_UNCHAINED_TRANSACTIONS and the conversation is included in the current transaction))	
CMPTR - Prepare_To_Receive	(see notes 1, 2 and 4 at end of table.)
if: *prepare_to_receive_type* = CM_PREP_TO_RECEIVE_SYNC_LEVEL **and** *sync_level* = CM_SYNC_POINT **and** (*transaction_control* = CM_UNCHAINED_TRANSACTIONS and the conversation is not currently included in the transaction)	
CMPTR - Prepare_To_Receive confirmation_urgency (characteristic)	TP-HANDSHAKE-AND-GRANT-CONTROL request Confirmation-Urgency
if: *prepare_to_receive_type* = CM_PREP_TO_RECEIVE_SYNC_LEVEL **and** *sync_level* = CM_SYNC_POINT_NO_CONFIRM **and** (*transaction_control* = CM_UNCHAINED_TRANSACTIONS and the conversation is not currently included in the transaction)	
CMPTR - Prepare_To_Receive	(see note 3 at end of table.) TP-GRANT-CONTROL request (no parameters)
if: *prepare_to_receive_type* = CM_PREP_TO_RECEIVE_FLUSH	
CMPTR - Prepare_To_Receive	(see notes 2 and 3 at end of table.) TP-GRANT-CONTROL request (no parameters)

CPI-C Call Parameters and Conversation Characteristics	**OSI TP Service Parameters**
if: *prepare_to_receive_type* = CM_PREP_TO_RECEIVE_CONFIRM	
CMPTR - Prepare_To_Receive	(see note 2 at end of table.) TP-HANDSHAKE-AND-GRANT-CONTROL request
confirmation_urgency (characteristic)	Confirmation-Urgency
Note 1: The completion of the mapping of Prepare_To_Receive depends on the following operation on the conversation. If the operation is: Flush — a TP-GRANT-CONTROL request will be sent Confirm — a TP-HANDSHAKE-AND-GRANT-CONTROL request will be sent a commit call — a TP-DEFERRED-GRANT-CONTROL request will be sent Prepare — a TP-DEFERRED-GRANT-CONTROL request will be sent.	
Note 2: if *sync_level* = CM_SYNC_POINT_NO_CONFIRM **and** *transaction_control* = CM_UNCHAINED_TRANSACTIONS **and** *begin_transaction* = CM_BEGIN_IMPLICIT **and** the Prepare_To_Receive call is the first activity on the conversation following the start of the current transaction, **then** the identified OSI TP service primitives are preceded by a TP-BEGIN-TRANSACTION request.	
Note 3: if: a TP-U-ERROR indication has arrived but has not been indicated to the program **then:** do not send any OSI TP Service primitive.	
Note 4: if: a TP-U-ERROR indication has arrived but has not been indicated to the program **then:** do not send any OSI TP Service primitive, a subsequent *tx_commit*() will generate a TP-ROLLBACK request.	

Table D-12 CMRTS — Request_To_Send Mapping

CPI-C Call Parameters and Conversation Characteristics	OSI TP Service Parameters
CMRTS - Request_To_Send	TP-REQUEST-CONTROL request (no parameters)

Table D-13 CMRCV — Receive

CPI-C Call Parameters and Conversation Characteristics	OSI TP Service Parameters
if: Confirmation = true on TP-BEGIN-DIALOGUE indication **and** it is the first activity on the conversation following Accept_Conversation or Accept_Incoming	
CMRCV - Receive = accepted *initialization_data* (characteristic)	TP-BEGIN-DIALOGUE response Result User-Data
if: *conversation_state* = CM_SEND_STATE or CM_SEND_PENDING_STATE **and** *sync_level* = CM_SYNC_POINT or *sync_level* = CM_SYNC_POINT_NO_CONFIRM **and** *transaction_control* = CM_UNCHAINED_TRANSACTIONS **and** *begin_transaction* = CM_BEGIN_IMPLICIT **and** it is the first activity on the conversation following the start of the current transaction	
CMRCV - Receive	TP-BEGIN-TRANSACTION request (no parameters) TP-GRANT-CONTROL request (no parameters)
if: *conversation_state* = CM_SEND_STATE	
CMRCV - Receive	TP-GRANT-CONTROL request (no parameters)

Table D-14 CMSEND — Send_Data Mapping

CPI-C Call Parameters and Conversation Characteristics	OSI TP Service Parameters
if: *send_type* = CM_BUFFER_DATA	
CMSEND - Send_Data *buffer* (parameter)	(See note 1 at end of table.) (TP-DATA request) UD-TRANSFER request User-Data
if: *send_type* = CM_SEND_AND_FLUSH	
CMSEND - Send_Data *buffer* (parameter)	(See note 1 at end of table.) TP-DATA request) UD-TRANSFER request User-Data
if: *send_type* = CM_SEND_AND_CONFIRM	
CMSEND - Send_Data *buffer* (parameter) *confirmation_urgency* (characteristic)	(See note 1 at end of table.) (TP-DATA request) UD-TRANSFER request User-Data TP-HANDSHAKE request Confirmation-Urgency
if: *send_type* = CM_SEND_AND_PREP_TO_RECEIVE **and** *prepare_to_receive_type* = CM_PREP_TO_RECEIVE_SYNC_LEVEL **and** *sync_level* = CM_NONE	
CMSEND - Send_Data *buffer* (parameter)	(TP-DATA request) UD-TRANSFER request User-Data TP-GRANT-CONTROL request (no parameters)
if: *send_type* = CM_SEND_AND_PREP_TO_RECEIVE **and** *prepare_to_receive_type* = CM_PREP_TO_RECEIVE_SYNC_LEVEL **and** *sync_level* = CM_CONFIRM	
CMSEND - Send_Data *buffer* (parameter) *confirmation_urgency* (characteristic)	TP-DATA request) UD-TRANSFER request User-Data TP-HANDSHAKE-AND-GRANT-CONTROL request Confirmation-Urgency
if: *send_type* = CM_SEND_AND_PREP_TO_RECEIVE **and** *prepare_to_receive_type* = CM_PREP_TO_RECEIVE_SYNC_LEVEL **and** *sync_level* = CM_SYNC_POINT or *sync_level* = CM_SYNC_POINT_NO_CONFIRM **and** (*transaction_control* = CM_CHAINED_TRANSACTIONS or (*transaction_control* = CM_UNCHAINED_TRANSACTIONS and the conversation is included in the current transaction))	
CMSEND - Send_Data *buffer* (parameter)	(See note 1 at end of table.) (TP-DATA request) UD-TRANSFER request User-Data (See note 2 at end of table.)

CPI-C Call Parameters and Conversation Characteristics	**OSI TP Service Parameters**
if: *send_type* = CM_SEND_AND_PREP_TO_RECEIVE **and** *prepare_to_receive_type* = CM_PREP_TO_RECEIVE_SYNC_LEVEL **and** *sync_level* = CM_SYNC_POINT **and** (*transaction_control* = CM_UNCHAINED_TRANSACTIONS and the conversation is not currently included in the transaction)	
CMSEND - Send_Data *buffer* (parameter) *confirmation_urgency* (characteristic)	(TP-DATA request) UD-TRANSFER request User-Data TP-HANDSHAKE-AND-GRANT-CONTROL request Confirmation-Urgency
if: *send_type* = CM_SEND_AND_PREP_TO_RECEIVE **and** *prepare_to_receive_type* = CM_PREP_TO_RECEIVE_SYNC_LEVEL **and** *sync_level* = CM_SYNC_POINT_NO_CONFIRM **and** (*transaction_control* = CM_UNCHAINED_TRANSACTIONS and the conversation is not currently included in the transaction)	
CMSEND - Send_Data *buffer* (parameter)	(TP-DATA request) UD-TRANSFER request User-Data TP-GRANT-CONTROL request (no parameters)
if: *send_type* = CM_SEND_AND_PREP_TO_RECEIVE **and** *prepare_to_receive_type* = CM_PREP_TO_RECEIVE_FLUSH	
CMSEND - Send_Data *buffer* (parameter)	(See note 1 at end of table.) (TP-DATA request) UD-TRANSFER request User-Data TP-GRANT-CONTROL request (no parameters)
if: *send_type* = CM_SEND_AND_PREP_TO_RECEIVE **and** *prepare_to_receive_type* = CM_PREP_TO_RECEIVE_CONFIRM	
CMSEND - Send_Data *buffer* (parameter) *confirmation_urgency* (characteristic)	(See note 1 at end of table.) (TP-DATA request) UD-TRANSFER request User-Data TP-HANDSHAKE-AND-GRANT-CONTROL request Confirmation-Urgency
if: *send_type* = CM_SEND_AND_DEALLOCATE **and** *deallocate_type* = CM_DEALLOCATE_SYNC_LEVEL **and** *sync_level* = CM_NONE	
CMSEND - Send_Data *buffer* (parameter) =false	(TP-DATA request) UD-TRANSFER request User-Data TP-END-DIALOGUE request Confirmation
if: *send_type* = CM_SEND_AND_DEALLOCATE **and** *deallocate_type* = CM_DEALLOCATE_SYNC_LEVEL **and** *sync_level* = CM_CONFIRM	
CMSEND - Send_Data *buffer* (parameter) =true	(TP-DATA request) UD-TRANSFER request User-Data TP-END-DIALOGUE request Confirmation

CPI-C Call Parameters and Conversation Characteristics	**OSI TP Service Parameters**
if: *send_type* = CM_SEND_AND_DEALLOCATE **and** *deallocate_type* = CM_DEALLOCATE_SYNC_LEVEL **and** *sync_level* = CM_SYNC_POINT or *sync_level* = CM_SYNC_POINT_NO_CONFIRM **and** (*transaction_control* = CM_CHAINED_TRANSACTIONS or (*transaction_control* = CM_UNCHAINED_TRANSACTIONS and the conversation is included in the current transaction))	
CMSEND - Send_Data *buffer* (parameter)	(See note 1 at end of table.) (TP-DATA request) UD-TRANSFER request User-Data TP-DEFERRED-END-DIALOGUE request (no parameters)
if: *send_type* = CM_SEND_AND_DEALLOCATE **and** *deallocate_type* = CM_DEALLOCATE_SYNC_LEVEL **and** *sync_level* = CM_SYNC_POINT **and** (*transaction_control* = CM_UNCHAINED_TRANSACTIONS and the conversation is not currently included in the transaction)	
CMSEND - Send_Data *buffer* (parameter) =true	(TP-DATA request) UD-TRANSFER request User-Data TP-END-DIALOGUE request Confirmation
if: *send_type* = CM_SEND_AND_DEALLOCATE **and** *deallocate_type* = CM_DEALLOCATE_SYNC_LEVEL **and** *sync_level* = CM_SYNC_POINT_NO_CONFIRM **and** (*transaction_control* = CM_UNCHAINED_TRANSACTIONS and the conversation is not currently included in the transaction)	
CMSEND - Send_Data *buffer* (parameter) =false	(TP-DATA request) UD-TRANSFER request User-Data TP-END-DIALOGUE request Confirmation
if: *send_type* = CM_SEND_AND_DEALLOCATE **and** *deallocate_type* = CM_DEALLOCATE_FLUSH	
CMSEND - Send_Data *buffer* (parameter) =false	(TP-DATA request) UD-TRANSFER request User-Data TP-END-DIALOGUE request Confirmation
if: *send_type* = CM_SEND_AND_DEALLOCATE **and** *deallocate_type* = CM_DEALLOCATE_CONFIRM	
CMSEND - Send_Data *buffer* (parameter) =true	(TP-DATA request) UD-TRANSFER request User-Data TP-END-DIALOGUE request Confirmation

CPI-C Call Parameters and Conversation Characteristics	**OSI TP Service Parameters**
if: *send_type* = CM_SEND_AND_DEALLOCATE **and** *deallocate_type* = CM_DEALLOCATE_ABEND	
CMSEND - Send_Data	(See note 1 at end of table.)
buffer (parameter)	(TP-DATA request) UD-TRANSFER request User-Data
log_data (characteristic)	TP-U-ABORT request User-Data
Note 1: if *sync_level* = CM_SYNC_POINT_NO_CONFIRM **and** *transaction_control* = CM_UNCHAINED_TRANSACTIONS **and** *begin_transaction* = CM_BEGIN_IMPLICIT **and** the Send_Data call is the first activity on the conversation following the start of the current transaction, **then** the identified OSI TP service primitives are preceded by a TP-BEGIN-TRANSACTION request.	
Note 2: The completion of the mapping of Send_Data depends on the following operation on the conversation. If the operation is: Flush — a TP-GRANT-CONTROL request will be sent Confirm — a TP-HANDSHAKE-AND-GRANT-CONTROL request will be sent a commit call — a TP-DEFERRED-GRANT-CONTROL request will be sent Prepare — a TP-DEFERRED-GRANT-CONTROL request will be sent.	
General Note: The data specified by the *buffer* and *send_length* parameters of the CMSEND call maps exactly to the User-Data of the UD-TRANSFER request, independent of the value of *conversation_type*.	

Table D-15 CMSERR — Send_Error Mapping

CPI-C Call Parameters and Conversation Characteristics	**OSI TP Service Parameters**
if: Confirmation = true on TP-BEGIN-DIALOGUE indication **and** it is the first activity on the conversation following Accept_Conversation or Accept_Incoming	
CMSERR - Send_Error = accepted *initialization_data* (characteristic)	TP-BEGIN-DIALOGUE response Result User-Data TP-U-ERROR request (no parameters)
if: *sync_level* = CM_SYNC_POINT or *sync_level* = CM_SYNC_POINT_NO_CONFIRM **and** *transaction_control* = CM_UNCHAINED_TRANSACTIONS **and** *begin_transaction* = CM_BEGIN_IMPLICIT **and** it is the first activity on the conversation following the start of the current transaction	
CMSERR - Send_Error	TP-BEGIN-TRANSACTION request (no parameters) TP-U-ERROR request (no parameters)
if: the local CRM has received a TP-Prepare indication	
CMSERR - Send_Error	TP-ROLLBACK request (no parameters)
otherwise:	
CMSERR - Send_Error	TP-U-ERROR request (no parameters)

D.1.3 Mapping OSI TP Services to CPI-C for Half-duplex Conversations

The following tables present the complete mapping from the OSI TP services to CPI-C for half-duplex conversations.

The following conventions are used within the tables in this section:

- The parameter is not applicable because of other parameter settings.

* The parameter is not directly supported by CPI-C.

=xxx The value xxx is expected by CPI-C.

Table D-16 Mapping OSI TP to CPI-C Calls, Parameters and Characteristics

OSI TP Service	CPI-C Calls, Parameters and Characteristics
(TP-DATA indication) UD-TRANSFER indication	Completes an outstanding CMRCV - Receive.
TP-BEGIN-DIALOGUE indication (See Table D-17 on page 445 for details.)	Completes an outstanding CMACCP - Accept_Conversation CMACCI - Accept_Incoming and maps to conversation characteristics.
TP-BEGIN-DIALOGUE confirm (See Table D-18 on page 446 for details.)	Completes an outstanding CMRCV - Receive and maps to *control_information_received* values.
TP-END-DIALOGUE indication (See Table D-19 on page 448 for details.)	Completes an outstanding CMRCV - Receive and maps to *status_received* or *return_code* values.
TP-END-DIALOGUE confirm (no parameters)	Completes an outstanding CMSEND - Send_Data CMDEAL - Deallocate and maps to *return_code* = CM_OK.
TP-U-ERROR indication (no parameters)	Completes an outstanding CMCFM - Confirm CMDEAL - Deallocate CMDFDE - Deferred_Deallocate CMINCL - Include_Partner_In_Transaction CMPREP - Prepare CMPTR - Prepare_To_Receive CMRCV - Receive CMSEND - Send_Data CMSERR - Send_Error and maps to *return_code* = CM_PROGRAM_ERROR_PURGING.
TP-U-ABORT indication (See Table D-20 on page 449 for details.)	Completes an outstanding CMRCV - Receive and maps to *return_code* values.
TP-P-ABORT indication (See Table D-21 on page 450 for details.)	Completes an outstanding CMRCV - Receive and maps to *return_code* values.
TP-GRANT-CONTROL indication (no parameters)	Completes an outstanding CMRCV - Receive and maps to *status_received* = CM_SEND_RECEIVED.
TP-REQUEST-CONTROL indication (no parameters)	Maps to *control_information_received* = CM_REQ_TO_SEND_RECEIVED.
TP-HANDSHAKE indication (no parameters)	Completes an outstanding CMRCV - Receive and maps to *status_received* = CM_CONFIRM_RECEIVED.

OSI TP Service	CPI-C Calls, Parameters and Characteristics
TP-HANDSHAKE confirm (no parameters)	Completes an outstanding CMCFM - Confirm CMSEND - Send_Data and maps to *return_code* = CM_OK.
TP-HANDSHAKE-AND-GRANT-CONTROL indication (no parameters)	Completes an outstanding CMRCV - Receive and maps to *status_received* = CM_CONFIRM_SEND_RECEIVED.
TP-HANDSHAKE-AND-GRANT-CONTROL confirm (no parameters)	Completes an outstanding CMPTR - Prepare_To_Receive CMSEND - Send_Data and maps to *return_code* = CM_OK.
TP-BEGIN-TRANSACTION indication (no parameters)	Completes an outstanding CMRCV - Receive and maps to *status_received* = CM_JOIN_TRANSACTION.
TP-DEFERRED-END-DIALOGUE indication (See Table D-22 on page 451 for details.)	Maps to *status_received* values.
TP-DEFERRED-GRANT-CONTROL indication (See Table D-22 on page 451 for details.)	Maps to *status_received* values.
TP-PREPARE indication (See Table D-22 on page 451 for details.)	Completes an outstanding CMRCV - Receive and maps to *status_received* values.
TP-READY indication	Completes an outstanding CMRCV - Receive and maps to *status_received* = CM_PREPARE_OK.
TP-COMMIT indication	(Handled by a resource recovery component.)
TP-COMMIT-COMPLETE indication	(Handled by a resource recovery component.)
TP-ROLLBACK indication (no parameters)	Maps to *return_code* = CM_TAKE_BACKOUT.
TP-ROLLBACK-COMPLETE indication	(Handled by a resource recovery component.)
TP-HEURISTIC-REPORT indication	(Handled by a resource recovery component.)

Table D-17 TP-BEGIN-DIALOGUE indication Mapping

OSI TP Service Parameters	CPI-C Calls, Parameters and Characteristics
TP-BEGIN-DIALOGUE indication	Maps to conversation characteristics:
Initiating-AP-Title	*AP_title* (characteristic)
Initiating-API-Identifier	*
Initiating-AE-Qualifier	*AE_qualifier* (characteristic)
Initiating-AEI-Identifier	*
Recipient-TPSU-Title	*TP_name* (characteristic)
Functional-Units	*sync_level*, *send_receive_mode*, and *transaction_control* (characteristics) and TX *transaction_control* characteristic
Begin-Transaction	
true	Completes an outstanding CMRCV - Receive and maps to status_received = CM_JOIN_TRANSACTION (also available through a resource recovery interface)
false	* (through a resource recovery interface)
Confirmation	* (confirmation is implicit)
User-Data	*initialization_data* (characteristic)

Table D-18 TP-BEGIN-DIALOGUE confirm Mapping

OSI TP Service Parameters	CPI-C Calls, Parameters and Characteristics
if: Result = accepted **and** User-Data is not present	
TP-BEGIN-DIALOGUE confirm	Maps to *control_information_received* = CM_ALLOCATE_CONFIRMED.
Functional-Units	*
Diagnostic	-
Rollback	-
User-Data	-
if: Result = accepted **and** User-Data is present	
TP-BEGIN-DIALOGUE confirm	Maps to *control_information_received* = CM_ALLOCATE_CONFIRMED_WITH_DATA
Functional-Units	*
Diagnostic	-
Rollback	-
User-Data	*initialization_data* (characteristic)
if: Result = rejected(user) **and** User-Data is not present	
TP-BEGIN-DIALOGUE confirm	Maps to *return_code* values.
Functional-Units	*
Diagnostic	-
Rollback	
true	CM_DEALLOCATED_ABEND_BO
false	CM_DEALLOCATED_ABEND
User-Data	*initialization_data* (characteristic)
if: Result = rejected(user) **and** User-Data is present	
TP-BEGIN-DIALOGUE confirm	Maps to *control_information_received* = CM_ALLOCATE_REJECTED_WITH_DATA
Functional-Units	*
Diagnostic	-
Rollback	-
User-Data	*initialization_data* (characteristic) **Note:** CM_DEALLOCATED_ABEND will be returned to the next call.

OSI TP Service Parameters	CPI-C Calls, Parameters and Characteristics
if: Result = rejected(provider) **and** Rollback = true	
TP-BEGIN-DIALOGUE confirm	Maps to *return_code* values.
Functional-Units	*
Diagnostic	
recipient-unknown	CM_RESOURCE_FAIL_NO_RETRY_BO
recipient-tpsu-title-unknown	CM_TPN_NOT_RECOGNIZED
tpsu-not-available(permanent)	CM_TP_NOT_AVAILABLE_NO_RETRY
tpsu-not-available(transient)	CM_TP_NOT_AVAILABLE_RETRY
recipient-tpsu-title-required	CM_TPN_NOT_RECOGNIZED
functional-unit-not-supported	if commit, chained transactions or unchained transactions functional units were requested return CM_SYNC_LVL_NOT_SUPPORTED_SYS, otherwise, if handshake functional unit was requested return CM_SYNC_LVL_NOT_SUPPORTED_PGM, otherwise, return CM_SEND_RCV_MODE_NOT_SUPPORTED.
functional-unit-combination-not-supported	if commit, chained transactions or unchained transactions functional units were requested return CM_SYNC_LVL_NOT_SUPPORTED_SYS, otherwise, if handshake functional unit was requested return CM_SYNC_LVL_NOT_SUPPORTED_PGM, otherwise, return CM_SEND_RCV_MODE_NOT_SUPPORTED.
no-reason-given	CM_RESOURCE_FAIL_NO_RETRY_BO
User-Data	-
if: Result = rejected(provider) **and** Rollback = false	
TP-BEGIN-DIALOGUE confirm	Maps to *return_code* values.
Functional-Units	*
Diagnostic	
recipient-unknown	CM_RESOURCE_FAILURE_NO_RETRY
recipient-tpsu-title-unknown	CM_TPN_NOT_RECOGNIZED
tpsu-not-available(permanent)	CM_TP_NOT_AVAILABLE_NO_RETRY
tpsu-not-available(transient)	CM_TP_NOT_AVAILABLE_RETRY
recipient-tpsu-title-required	CM_TPN_NOT_RECOGNIZED
functional-unit-not-supported	CM_SYNC_LVL_NOT_SUPPORTED_SYS
functional-unit-combination-not-supported	CM_SYNC_LVL_NOT_SUPPORTED_SYS
no-reason-given	CM_RESOURCE_FAILURE_NO_RETRY
User-Data	-

Table D-19 TP-END-DIALOGUE indication Mapping

OSI TP Service Parameters	CPI-C Calls, Parameters and Characteristics
if: Confirmation = true	
TP-END-DIALOGUE indication (no parameters)	*status_received* = CM_CONFIRM_DEALLOC_RECEIVED
if: Confirmation = false	
TP-END-DIALOGUE indication (no parameters)	*return_code* = CM_DEALLOCATED_NORMAL

Table D-20 TP-U-ABORT indication Mapping

OSI TP Service Parameters	CPI-C Calls, Parameters and Characteristics
if: Rollback = true	
TP-U-ABORT indication User_Data	*return_code* = CM_DEALLOCATED_ABEND_BO *log_data* (characteristic)
if: Rollback = false	
TP-U-ABORT indication User_Data	*return_code* = CM_DEALLOCATED_ABEND *log_data* (characteristic)

Table D-21 TP-P-ABORT indication Mapping

OSI TP Service Parameters	CPI-C Calls, Parameters and Characteristics
if: Rollback = true	
TP-P-ABORT indication	Maps to *return_code* values:
Diagnostic	
permanent-failure	CM_RESOURCE_FAIL_NO_RETRY_BO
transient-failure	CM_RESOURCE_FAILURE_RETRY_BO
protocol-error	CM_RESOURCE_FAIL_NO_RETRY_BO
begin-transaction-reject	*
end-dialogue-collision	*
begin-transaction-end-dialogue-collision	*
if: Rollback = false	
TP-P-ABORT indication	Maps to *return_code* values:
Diagnostic	
permanent-failure	CM_RESOURCE_FAILURE_NO_RETRY
transient-failure	CM_RESOURCE_FAILURE_RETRY
protocol-error	CM_RESOURCE_FAILURE_NO_RETRY
begin-transaction-reject	*
end-dialogue-collision	*
begin-transaction-end-dialogue-collision	*

Table D-22 TP-DEFERRED-* and TP-PREPARE indication Mapping

OSI TP Service Parameters	CPI-C Calls, Parameters and Characteristics
if: a TP-DEFERRED-END-DIALOGUE indication has been received	
TP-PREPARE indication	Maps to *Status_received* values:
Data-Permitted	
false	CM_TAKE_COMMIT_DEALLOCATE
true	CM_TAKE_COMMIT_DEALLOC_DATA_OK
if: a TP-DEFERRED-END-DIALOGUE indication has NOT been received **and** a TP-DEFERRED-GRANT-CONTROL indication has been received	
TP-PREPARE indication	Maps to *Status_received* values:
Data-Permitted	
false	CM_TAKE_COMMIT_SEND
true	CM_TAKE_COMMIT_SEND_DATA_OK
if: a TP-DEFERRED-END-DIALOGUE indication has NOT been received **and** a TP-DEFERRED-GRANT-CONTROL indication has NOT been received	
TP-PREPARE indication	Maps to *Status_received* values:
Data-Permitted	
false	CM_TAKE_COMMIT
true	CM_TAKE_COMMIT_DATA_OK

D.1.4 Sequencing Rules and State Tables

The sequencing rules and state tables for the CPI-C ASE are contained in Appendix C.

D.1.5 CPI-C ASE Protocol Definition

There are no additional protocol definitions required by the CPI-C ASE beyond those defined in ISO/IEC 10026, Parts 1-6.

D.1.6 CPI-C ASE Structure and Encoding of APDUs

There are no additional APDUs required by the CPI-C ASE beyond those defined in ISO/IEC 10026, Parts 1-6.

D.2 OSI TP CRMs (Full-duplex)

This section summarizes the CPI-C application service element (ASE) services, maps the services both to and from the OSI TP services, and defines the sequencing rules and state table for CPI-C use by OSI TP programs using full-duplex conversations.

The CPI Communications calls have been mapped to the OSI TP services described in ISO/IEC 10026-2, Information Technology — Open Systems Interconnection — Distributed Transaction Processing — Part 2: OSI TP Service Definition.

This section is intended for programmers who are familiar with OSI TP.

The following conventions are used within the tables in this section:

* The parameter is not directly supported by CPI-C.

=xxx The value xxx is always used.

(R) returned on the Receive call

(S) returned on calls associated with the Send queue

The abbreviations for *status_received* and return code values used in this mapping are given in Section C.6.3 on page 404 and Section C.6.4 on page 407, respectively.

Table D-23 Mapping CPI-C Calls on Full-duplex Conversations to OSI TP

CPI-C Call	OSI TP Service
CMALLC - Allocate (See Table D-24 on page 455 for details.)	TP-BEGIN-DIALOGUE request
CMCANC - Cancel_Conversation (See Table D-25 on page 457 for details.)	TP-BEGIN-DIALOGUE response TP-U-ABORT request
CMCFMD - Confirmed (See Table D-26 on page 458 for details.)	TP-END-DIALOGUE response
CMDEAL - Deallocate (See Table D-27 on page 459 for details.)	TP-END-DIALOGUE request TP-DEFERRED-END-DIALOGUE request TP-BEGIN-DIALOGUE response TP-U-ABORT request
CMDFDE - Deferred_Deallocate (See Table D-28 on page 461 for details.)	TP-DEFERRED-END-DIALOGUE request
CMINCL - Include_Partner_In_Transaction (See Table D-29 on page 462 for details.)	TP-BEGIN-TRANSACTION request
CMPREP - Prepare (See Table D-30 on page 463 for details.)	TP-BEGIN-TRANSACTION request TP-PREPARE request
CMRCV - Receive (See Table D-31 on page 464 for details.)	TP-BEGIN-DIALOGUE response TP-BEGIN-TRANSACTION request
CMSEND - Send_Data (See Table D-32 on page 465 for details.)	TP-BEGIN-DIALOGUE response TP-BEGIN-TRANSACTION request (TP-DATA request) UD-TRANSFER request TP-END-DIALOGUE request TP-DEFERRED-END-DIALOGUE request TP-U-ABORT request
CMSERR - Send_Error (See Table D-33 on page 468 for details.)	TP-BEGIN-DIALOGUE response TP-BEGIN-TRANSACTION request TP-U-ERROR request

The following OSI TP services are not directly mapped to from the CPI-C calls:

TP-COMMIT request
: is supported through a resource recovery interface.

TP-DONE request
: is not externalized to the application program.

TP-ROLLBACK request
: is supported through a resource recovery interface.

Table D-24 CMALLC — Allocate Mapping

CPI-C Call Parameters and Conversation Characteristics	OSI TP Service Parameters
if: *sync_level* = CM_NONE	
CMALLC - Allocate	TP-BEGIN-DIALOGUE request
*	Initiating-TPSU-Title
AP_title (characteristic)	Recipient-AP-Title
*	Recipient-API-Identifier
AE_qualifier (characteristic)	Recipient-AE-Qualifier
*	Recipient-AEI-Identifier
TP_name (characteristic)	Recipient-TPSU-Title
=Dialogue and Shared	Functional-Units
mode_name (characteristic)	Quality-of-Service
application_context_name (characteristic)	Application-Context-Name
n/a	Begin-Transaction
allocate_confirm (characteristic)	Confirmation
initialization_data (characteristic)	User-Data
if: *sync_level* = CM_SYNC_POINT_NO_CONFIRM **and** *transaction_control* = CM_CHAINED_TRANSACTIONS	
CMALLC - Allocate	TP-BEGIN-DIALOGUE request
*	Initiating-TPSU-Title
AP_title (characteristic)	Recipient-AP-Title
*	Recipient-API-Identifier
AE_qualifier (characteristic)	Recipient-AE-Qualifier
*	Recipient-AEI-Identifier
TP_name (characteristic)	Recipient-TPSU-Title
=Dialogue, Shared, Commit and Chained Transactions	Functional-Units
mode_name (characteristic)	Quality-of-Service
application_context_name (characteristic)	Application-Context-Name
n/a	Begin-Transaction
allocate_confirm (characteristic)	Confirmation
initialization_data (characteristic)	User-Data

CPI-C Call Parameters and Conversation Characteristics	**OSI TP Service Parameters**
if: *sync_level* = CM_SYNC_POINT_NO_CONFIRM **and** *transaction_control* = CM_UNCHAINED_TRANSACTIONS	
CMALLC - Allocate	TP-BEGIN-DIALOGUE request
*	Initiating-TPSU-Title
AP_title (characteristic)	Recipient-AP-Title
*	Recipient-API-Identifier
AE_qualifier (characteristic)	Recipient-AE-Qualifier
*	Recipient-AEI-Identifier
TP_name (characteristic)	Recipient-TPSU-Title
=Dialogue, Shared, Commit, and Unchained Transactions	Functional-Units
mode_name (characteristic)	Quality-of-Service
application_context_name (characteristic)	Application-Context-Name
begin_transaction (characteristic)	Begin-Transaction (See note at end of table.)
allocate_confirm (characteristic)	Confirmation
initialization_data (characteristic)	User-Data
Note: if: begin_transaction = CM_BEGIN_IMPLICIT **and** *transaction_control* = CM_UNCHAINED_TRANSACTIONS **and** the program is in transaction mode **then** Begin-Transaction is set to true; **otherwise:** Begin-Transaction is set to false.	

Table D-25 CMCANC — Cancel_Conversation Mapping

CPI-C Call Parameters and Conversation Characteristics	OSI TP Service Parameters
if: Confirmation = true on TP-BEGIN-DIALOGUE indication **and** it is the first activity on the conversation following Accept_Conversation or Accept_Incoming	
CMCANC - Cancel_Conversation = rejected(user) *initialization_data* (characteristic)	TP-BEGIN-DIALOGUE response Result User-Data
otherwise:	
CMCANC - Cancel_Conversation =null	TP-U-ABORT request User-Data

Table D-26 CMCFMD — Confirmed Mapping

CPI-C Call Parameters and Conversation Characteristics	OSI TP Service Parameters
if in response to: *status_received* = CM_CONFIRM_DEALLOC_RECEIVED	
CMCFMD - Confirmed	TP-END-DIALOGUE response (no parameters)

Table D-27 CMDEAL — Deallocate Mapping

CPI-C Call Parameters and Conversation Characteristics	**OSI TP Service Parameters**
if: *deallocate_type* = CM_DEALLOCATE_SYNC_LEVEL **and** *sync_level* = CM_NONE	
CMDEAL - Deallocate =false =false	(See note at end of table.) TP-END-DIALOGUE request Confirmation (and TP-END-DIALOGUE indication Confirmation **if** (TP-END-DIALOGUE or TP-*-ABORT indication) has not been processed)
if: *deallocate_type* = CM_DEALLOCATE_SYNC_LEVEL **and** *sync_level* = CM_SYNC_POINT_NO_CONFIRM **and** (*transaction_control* = CM_CHAINED_TRANSACTIONS **or** (*transaction_control* = CM_UNCHAINED_TRANSACTIONS **and** conversation is included in a transaction))	
CMDEAL - Deallocate	(See note at end of table.) TP-DEFERRED-END-DIALOGUE request (no parameters)
if: *deallocate_type* = CM_DEALLOCATE_SYNC_LEVEL **and** *sync_level* = CM_SYNC_POINT_NO_CONFIRM **and NOT** (*transaction_control* = CM_CHAINED_TRANSACTIONS **or** (*transaction_control* = CM_UNCHAINED_TRANSACTIONS **and** conversation is included in a transaction))	
CMDEAL - Deallocate =false =false	(See note at end of table.) TP-END-DIALOGUE request Confirmation (and TP-END-DIALOGUE indication Confirmation **if** (TP-END-DIALOGUE or TP-*-ABORT indication) has not been processed)
if: *deallocate_type* = CM_DEALLOCATE_FLUSH	
CMDEAL - Deallocate =false =false	(See note at end of table.) TP-END-DIALOGUE request Confirmation (and TP-END-DIALOGUE indication Confirmation **if** (TP-END-DIALOGUE or TP-*-ABORT indication) has not been processed)

CPI-C Call Parameters and Conversation Characteristics	**OSI TP Service Parameters**
if: *deallocate_type* = CM_DEALLOCATE_CONFIRM	
CMDEAL - Deallocate =true	(See note at end of table.) **if** no collision with CM_ALLOCATION_ERROR, CM_DEALLOCATED_ABEND_*, CM_RESOURCE_FAILURE_*_RETRY or CM_DEALLOCATED_NORMAL, **or** (ok,(*,jt)) **or** (ok,(*,cd)) TP-END-DIALOGUE request Confirmation **otherwise** return, respectively, CM_DEALLOC_CONFIRM_REJECT (S), **or** TP-U-ABORT request (Rollback = true), CM_DEALLOCATED_ABEND.begin_transaction_collision (R) and CM_DEALLOCATED_ABEND.begin_transaction_collision (S), **or** TP-U-ABORT request (Rollback = false), CM_DEALLOCATED_ABEND.dealloc_confirm_collision (R) and CM_DEALLOCATED_ABEND.dealloc_confirm_collision (S). If collision results in a return code on the Receive call, replace the status_received value that caused the collision.
if: *deallocate_type* = CM_DEALLOCATE_ABEND **and** it is the first activity on the conversation following Accept_Conversation or Accept_Incoming	
CMDEAL - Deallocate =rejected(user) *initialization_data* (characteristic)	TP-BEGIN-DIALOGUE response Result User-Data
if: *deallocate_type* = CM_DEALLOCATE_ABEND **and** it is not the first activity on the conversation following Accept_Conversation or Accept_Incoming	
CMDEAL - Deallocate *log_data* (characteristic)	TP-U-ABORT request User-Data
Note: if Confirmation = true on TP-BEGIN-DIALOGUE indication **and** it is the first activity on the conversation following Accept_Conversation or Accept_Incoming **then** the identified OSI TP Service primitive will be preceded by TP-BEGIN-DIALOGUE response (Result = accepted and User-Data = *initialization_data* (characteristic)).	

Table D-28 CMDFDE — Deferred_Deallocate Mapping

CPI-C Call Parameters and Conversation Characteristics	OSI TP Service Parameters
CMDFDE - Deferred_Deallocate	TP-DEFERRED-END-DIALOGUE request (no parameters)

Table D-29 CMINCL — Include_Partner_In_Transaction Mapping

CPI-C Call Parameters and Conversation Characteristics	**OSI TP Service Parameters**
CMINCL - Include_Partner_In_Transaction	TP-BEGIN-TRANSACTION request (no parameters) if no collision with (ok,(*,dc)); **otherwise** return CM_DEALLOCATED_ABEND_BO.dealloc_confirm_collision (R), instead of (ok,(*,dc)), or CM_DEALLOCATED_ABEND_BO.dealloc_confirm_collision (S), and send TP-U-ABORT request (Rollback = false).

Table D-30 CMPREP — Prepare Mapping

CPI-C Call Parameters and Conversation Characteristics	**OSI TP Service Parameters**
if: *sync_level* = CM_SYNC_POINT_NO_CONFIRM **and** *transaction_control* = CM_UNCHAINED_TRANSACTIONS **and** *begin_transaction* = CM_BEGIN_IMPLICIT **and** it is the first activity on the conversation following the start of the current transaction	
CMPREP - Prepare	TP-BEGIN-TRANSACTION request (no parameters) TP-PREPARE request
prepare_data_permitted (characteristic)	Data-Permitted
if: *sync_level* = CM_SYNC_POINT_NO_CONFIRM **and not** (*transaction_control* = CM_UNCHAINED_TRANSACTIONS **and** *begin_transaction* = CM_BEGIN_IMPLICIT **and** it is the first activity on the conversation following the start of the current transaction)	
CMPREP - Prepare	TP-PREPARE request
prepare_data_permitted (characteristic)	Data-Permitted

Table D-31 CMRCV — Receive

CPI-C Call Parameters and Conversation Characteristics	OSI TP Service Parameters
if: Confirmation = true on TP-BEGIN-DIALOGUE indication **and** it is the first activity on the conversation following Accept_Conversation or Accept_Incoming	
CMRCV - Receive = accepted *initialization_data* (characteristic)	TP-BEGIN-DIALOGUE response Result User-Data
if: *sync_level* = CM_SYNC_POINT_NO_CONFIRM **and** *transaction_control* = CM_UNCHAINED_TRANSACTIONS **and** *begin_transaction* = CM_BEGIN_IMPLICIT **and** it is the first activity on the conversation following the start of the current transaction	
CMRCV - Receive	TP-BEGIN-TRANSACTION request (no parameters)

Table D-32 CMSEND — Send_Data Mapping

CPI-C Call Parameters and Conversation Characteristics	OSI TP Service Parameters
if: *send_type* = CM_BUFFER_DATA	
CMSEND - Send_Data *buffer* (parameter)	(See notes 1 and 2 at end of table.) (TP-DATA request) UD-TRANSFER request User-Data
if: *send_type* = CM_SEND_AND_FLUSH	
CMSEND - Send_Data *buffer* (parameter)	(See notes 1 and 2 at end of table.) (TP-DATA request) UD-TRANSFER request User-Data
if: *send_type* = CM_SEND_AND_DEALLOCATE **and** *deallocate_type* = CM_DEALLOCATE_SYNC_LEVEL **and** *sync_level* = CM_NONE	
CMSEND - Send_Data *buffer* (parameter) =false =false	(See note 1 at end of table.) (TP-DATA request) UD-TRANSFER request User-Data TP-END-DIALOGUE request Confirmation (and TP-END-DIALOGUE indication Confirmation **if** (TP-END-DIALOGUE or TP-*-ABORT indication) has not been processed)
if: *send_type* = CM_SEND_AND_DEALLOCATE **and** *deallocate_type* = CM_DEALLOCATE_SYNC_LEVEL **and** *sync_level* = CM_SYNC_POINT_NO_CONFIRM **and** ((*transaction_control* = CM_UNCHAINED_TRANSACTIONS **and not** (*begin_transaction* = CM_BEGIN_IMPLICIT **and** it is the first activity on the conversation following the start of the current transaction)) **or** *transaction_control* = CM_CHAINED_TRANSACTIONS) **and** the conversation is included in a transaction	
CMSEND - Send_Data *buffer* (parameter)	(See notes 1 and 2 at end of table.) (TP-DATA request) UD-TRANSFER request User-Data TP-DEFERRED-END-DIALOGUE request (no parameters)

CPI-C Call Parameters and Conversation Characteristics	**OSI TP Service Parameters**
if: *send_type* = CM_SEND_AND_DEALLOCATE **and** *deallocate_type* = CM_DEALLOCATE_SYNC_LEVEL **and** *sync_level* = CM_SYNC_POINT_NO_CONFIRM **and** ((*transaction_control* = CM_UNCHAINED_TRANSACTIONS **and not** (*begin_transaction* = CM_BEGIN_IMPLICIT **and** it is the first activity on the conversation following the start of the current transaction)) **or** *transaction_control* = CM_CHAINED_TRANSACTIONS) **and** the conversation is **not** currently included in a transaction	
	(See note 1 at end of table.)
CMSEND - Send_Data	(TP-DATA request) UD-TRANSFER request
buffer (parameter)	User-Data
	TP-END-DIALOGUE request
=false	Confirmation
	(and TP-END-DIALOGUE indication
=false	Confirmation
	if (TP-END-DIALOGUE or TP-*-ABORT indication) has not been processed)
if: *send_type* = CM_SEND_AND_DEALLOCATE **and** *deallocate_type* = CM_DEALLOCATE_FLUSH	
	(See note 1 at end of table.)
CMSEND - Send_Data	(TP-DATA request) UD-TRANSFER request
buffer (parameter)	User-Data
	TP-END-DIALOGUE request
=false	Confirmation
	(and TP-END-DIALOGUE indication
=false	Confirmation
	if (TP-END-DIALOGUE or TP-*-ABORT indication) has not been processed)

CPI-C Call Parameters and Conversation Characteristics	OSI TP Service Parameters
if: *send_type* = CM_SEND_AND_DEALLOCATE **and** *deallocate_type* = CM_DEALLOCATE_CONFIRM	
	(See note 1 at end of table.)
CMSEND - Send_Data	(TP-DATA request) UD-TRANSFER request
buffer (parameter)	User-Data.
	if no collision with CM_ALLOCATION_ERROR, CM_DEALLOCATED_ABEND_*, CM_RESOURCE_FAILURE_*_RETRY or CM_DEALLOCATED_NORMAL, **or** (ok,(*,jt)) **or** (ok,(*,cd))
	TP-END-DIALOGUE request
=true	Confirmation
	otherwise return, respectively, CM_DEALLOC_CONFIRM_REJECT (S), **or** TP-U-ABORT request (Rollback = true), CM_DEALLOCATED_ABEND.begin_transaction_collision (R) and CM_DEALLOCATED_ABEND.begin_transaction_collision (S), **or** TP-U-ABORT request (Rollback = false), CM_DEALLOCATED_ABEND.dealloc_confirm_collision (R) and CM_DEALLOCATED_ABEND.dealloc_confirm_collision (S). If collision results in a return code to the Receive call, replace the *status_received* value that caused the collision.
if: *send_type* = CM_SEND_AND_DEALLOCATE **and** *deallocate_type* = CM_DEALLOCATE_ABEND	
	(See notes 1 and 2 at end of table.)
CMSEND - Send_Data	(TP-DATA request) UD-TRANSFER request
buffer (parameter)	User-Data
	TP-U-ABORT request
log_data (characteristic)	User-Data
Note 1: if Confirmation = true on TP-BEGIN-DIALOGUE indication **and** it is the first activity on the conversation following Accept_Conversation or Accept_Incoming **then** the identified OSI TP Service primitive will be preceded by TP-BEGIN-DIALOGUE response (Result = accepted and User-Data = *initialization_data* (characteristic)).	
Note 2: if *sync_level* = CM_SYNC_POINT_NO_CONFIRM **and** *transaction_control* = CM_UNCHAINED_TRANSACTIONS **and** *begin_transaction* = CM_BEGIN_IMPLICIT **and** the Send_Data call is the first activity on the conversation following the start of the current transaction, **then** the identified OSI TP service primitives are preceded by a TP-BEGIN-TRANSACTION request.	
General Note: The data specified by the *buffer* and *send_length* parameters of the CMSEND call maps exactly to the User-Data of the UD-TRANSFER request, independent of the value of *conversation_type*.	

Table D-33 CMSERR — Send_Error Mapping

CPI-C Call Parameters and Conversation Characteristics	OSI TP Service Parameters
if: Confirmation = true on TP-BEGIN-DIALOGUE indication **and** it is the first activity on the conversation following Accept_Conversation or Accept_Incoming	
CMSERR - Send_Error = accepted *initialization_data* (characteristic)	TP-BEGIN-DIALOGUE response Result User-Data TP-U-ERROR request (no parameters)
if: *sync_level* = CM_SYNC_POINT_NO_CONFIRM **and** *transaction_control* = CM_UNCHAINED_TRANSACTIONS **and** *begin_transaction* = CM_BEGIN_IMPLICIT **and** it is the first activity on the conversation following the start of the current transaction	
CMSERR - Send_Error	TP-BEGIN-TRANSACTION request (no parameters) TP-U-ERROR request (no parameters)
otherwise:	
CMSERR - Send_Error	

D.2.1 Mapping OSI TP Services to CPI-C for Full-duplex Conversations

The following tables present the complete mapping from the OSI TP services to CPI-C for full-duplex conversations.

The following conventions are used within the tables in this section:

-	The parameter is not applicable because of other parameter settings.
*	The parameter is not directly supported by CPI-C.
=xxx	The value xxx is expected by CPI-C.
(R)	The return code is returned on the Receive call
(S)	The return code is returned on calls associated with Send queue
x (R) [+ y (S)]	If x is returned before y, then y need not be returned. If y is returned before x, then x must also be returned.
a.b	‘‘a’’ is the return code and ‘‘b’’ is the secondary information associated with the return code

Table D-34 Mapping OSI TP to CPI-C Calls, Parameters and Characteristics

OSI TP Service	CPI-C Calls, Parameters and Characteristics
(TP-DATA indication) UD-TRANSFER indication	Completes an outstanding CMRCV - Receive
TP-BEGIN-DIALOGUE indication (See Table D-17 on page 445 for details.)	Completes an outstanding CMACCP - Accept_Conversation CMACCI - Accept_Incoming and maps to conversation characteristics.
TP-BEGIN-DIALOGUE confirm (See Table D-18 on page 446 for details.)	Completes an outstanding CMRCV - Receive and maps to *control_information_received* values
TP-END-DIALOGUE indication (See Table D-19 on page 448 for details.)	Completes an outstanding CMRCV - Receive and maps to status_received or return_code values.
TP-END-DIALOGUE confirm (no parameters)	Completes an outstanding CMSEND - Send_Data CMDEAL - Deallocate and maps to *return_code* = CM_OK (S) and CM_DEALLOCATE_NORMAL (R)
TP-U-ERROR indication (no parameters)	Completes an outstanding CMRCV - Receive CMDEAL - Deallocate(Confirm) and maps to *return_code* = CM_PROGRAM_ERROR_PURGING (R) (**if** CMDEAL - DEALLOCATE(Confirm) is being processed, also maps to CM_DEALLOC_CONFIRM_REJECT (S))
TP-U-ABORT indication (See Table D-20 on page 449 for details.)	Completes an outstanding CMRCV - Receive and maps to *return_code* values.
TP-P-ABORT indication (See Table D-39 on page 477 for details.)	Completes an outstanding CMRCV - Receive CMDEAL - DEALLOCATE(Confirm) and maps to *return_code* values.

OSI TP Service	CPI-C Calls, Parameters and Characteristics
TP-BEGIN-TRANSACTION indication (no parameters)	**If** collision with outstanding CMDEAL(Confirm), **then** completes the outstanding CMDEAL - DEALLOCATE(Confirm) and maps to *return_code*= CM_DEALLOCATED_ABEND.begin_transaction_collision (R) and CM_DEALLOCATED_ABEND.begin_transaction_collision (S) and TP-U-ABORT request (Rollback = true) **else** if *conv_state* = SEND-RECEIVE **then** completes an outstanding CMRCV - Receive and maps to *status_received* = CM_JOIN_TRANSACTION
TP-DEFERRED-END-DIALOGUE indication (See Table D-22 on page 451 for details.)	Maps to *status_received* values.
TP-PREPARE indication (See Table D-22 on page 451 for details.)	Completes an outstanding CMRCV - Receive and maps to *status_received* values.
TP-READY indication	Completes an outstanding CMRCV - Receive and maps to *status_received* = CM_PREPARE_OK.
TP-COMMIT indication	(Handled by a resource recovery component.)
TP-COMMIT-COMPLETE indication	(Handled by a resource recovery component.)
TP-ROLLBACK indication (no parameters)	Maps to *return_code* = CM_TAKE_BACKOUT.
TP-ROLLBACK-COMPLETE indication	(Handled by a resource recovery component.)
TP-HEURISTIC-REPORT indication	(Handled by a resource recovery component.)

Table D-35 TP-BEGIN-DIALOGUE indication Mapping

OSI TP Service Parameters	CPI-C Calls, Parameters and Characteristics
TP-BEGIN-DIALOGUE indication	Maps to conversation characteristics:
Initiating-AP-Title	*AP_title* (characteristic)
Initiating-API-Identifier	*
Initiating-AE-Qualifier	*AE_qualifier* (characteristic)
Initiating-AEI-Identifier	*
Recipient-TPSU-Title	*TP_name* (characteristic)
Functional-Units	*sync_level*, *send_receive_mode*, and *transaction_control* (characteristics) and TX *transaction_control* characteristic **if** Handshake FU selected **then** TP-BEGIN-DIALOGUE response with Diagnostic=FU-comb-not-supported
Begin-Transaction	
true	Completes an outstanding CMRCV - Receive and maps to status_received = CM_JOIN_TRANSACTION (also available through a resource recovery interface)
false	* (through a resource recovery interface)
Confirmation	* (confirmation is implicit)
User-Data	*initialization_data* (characteristic)

Table D-36 TP-BEGIN-DIALOGUE confirm Mapping

OSI TP Service Parameters	CPI-C Calls, Parameters and Characteristics
if: Result = accepted **and** User-Data is not present	
TP-BEGIN-DIALOGUE confirm	Maps to *control_information_received* = CM_ALLOCATE_CONFIRMED.
Functional-Units	*
Diagnostic	-
Rollback	-
User-Data	-
if: Result = accepted **and** User-Data is present	
TP-BEGIN-DIALOGUE confirm	Maps to *control_information_received* = CM_ALLOCATE_CONFIRMED_WITH_DATA
Functional-Units	*
Diagnostic	-
Rollback	-
User-Data	*initialization_data* (characteristic)
if: Result = rejected(user) **and** User-Data is not present	
TP-BEGIN-DIALOGUE confirm	Maps to *return_code* values.
Functional-Units	*
Diagnostic	-
Rollback	
true	CM_DEALLOCATED_ABEND_BO (S or R)
false	CM_DEALLOCATED_ABEND (R) [+CM_DEALLOCATED_ABEND (S)]
User-Data	*initialization_data* (characteristic)
if: Result = rejected(user) **and** User-Data is present	
TP-BEGIN-DIALOGUE confirm	Maps to *control_information_received* = CM_ALLOCATE_REJECTED_WITH_DATA
Functional-Units	*
Diagnostic	-
Rollback	-
User-Data	*initialization_data* (characteristic) **Note:** CM_DEALLOCATED_ABEND will be returned to the next call.

OSI TP Service Parameters	CPI-C Calls, Parameters and Characteristics
if: Result = rejected(provider) **and** Rollback = true	
TP-BEGIN-DIALOGUE confirm	Maps to *return_code* values.
Functional-Units	*
Diagnostic	
recipient-unknown	CM_RESOURCE_FAIL_NO_RETRY_BO (S or R)
recipient-tpsu-title-unknown	CM_TPN_NOT_RECOGNIZED (R) [+ CM_ALLOCATION_ERROR (S)]
tpsu-not-available(permanent)	CM_TP_NOT_AVAILABLE_NO_RETRY (R) [+ CM_ALLOCATION_ERROR (S)]
tpsu-not-available(transient)	CM_TP_NOT_AVAILABLE_RETRY (R) [+ CM_ALLOCATION_ERROR (S)]
recipient-tpsu-title-required	CM_TPN_NOT_RECOGNIZED (R) [+ CM_ALLOCATION_ERROR (S)]
functional-unit-not-supported	if commit, chained transactions or unchained transactions functional units were requested return CM_SYNC_LVL_NOT_SUPPORTED_SYS (R), otherwise, if handshake functional unit was requested return CM_SYNC_LVL_NOT_SUPPORTED_PGM (R), otherwise, return CM_SEND_RCV_MODE_NOT_SUPPORTED (R) [+CM_ALLOCATION_ERROR (S)]
functional-unit-combination-not-supported	if commit, chained transactions or unchained transactions functional units were requested return CM_SYNC_LVL_NOT_SUPPORTED_SYS (S), otherwise, if handshake functional unit was requested return CM_SYNC_LVL_NOT_SUPPORTED_PGM (S), otherwise, return CM_SEND_RCV_MODE_NOT_SUPPORTED (S).
no-reason-given	CM_RESOURCE_FAIL_NO_RETRY_BO (S or R)
User-Data	-

OSI TP Service Parameters	CPI-C Calls, Parameters and Characteristics
if: Result = rejected(provider) **and** Rollback = false	
TP-BEGIN-DIALOGUE confirm	Maps to *return_code* values.
Functional-Units	*
Diagnostic	
recipient-unknown	CM_RESOURCE_FAILURE_NO_RETRY (R) [+ CM_RESOURCE_FAILURE_NO_RETRY (S)]
recipient-tpsu-title-unknown	CM_TPN_NOT_RECOGNIZED (R) [+ CM_ALLOCATION_ERROR (S)]
tpsu-not-available(permanent)	CM_TP_NOT_AVAILABLE_NO_RETRY (R) [+ CM_ALLOCATION_ERROR (S)]
tpsu-not-available(transient)	CM_TP_NOT_AVAILABLE_RETRY (R) [+ CM_ALLOCATION_ERROR (S)]
recipient-tpsu-title-required	CM_TPN_NOT_RECOGNIZED (R) [+ CM_ALLOCATION_ERROR (S)]
functional-unit-not-supported	CM_SYNC_LVL_NOT_SUPPORTED_SYS (R) or CM_SEND_RECEIVE_MODE_NOT_SUPPORTED (R), [+ CM_ALLOCATION_ERROR (S)]
functional-unit-combination-not-supported	CM_SYNC_LVL_NOT_SUPPORTED_SYS (R) and/or CM_SEND_RECEIVE_MODE_NOT_SUPPORTED (R), [+ CM_ALLOCATION_ERROR (S)]
no-reason-given	CM_RESOURCE_FAILURE_NO_RETRY (R) [+ CM_RESOURCE_FAILURE_NO_RETRY (S)]
User-Data	-

Table D-37 TP-END-DIALOGUE indication Mapping

OSI TP Service Parameters	CPI-C Calls, Parameters and Characteristics
if: Confirmation = true	
TP-END-DIALOGUE indication (no parameters)	**if** no collision with CMSERR **or** CMDEAL(C) **or** CMINCL, *status_received* = CM_CONFIRM_DEALLOC_RECEIVED **otherwise**, respectively, no-op **or** CM_DEALLOCATED_ABEND.dealloc_confirm_collision (R) [+ CM_DEALLOCATED_ABEND.dealloc_confirm_collision (S)], and TP-U-ABORT request (Rollback = false) **or** CM_DEALLOCATED_ABEND_BO.dealloc_confirm_collision (S or R) and TP-U-ABORT request (Rollback = false).
if: Confirmation = false	
TP-END-DIALOGUE indication (no parameters)	*return_code* = CM_DEALLOCATED_NORMAL (R) [+ CM_DEALLOCATED_NORMAL (S)]

Table D-38 TP-U-ABORT indication Mapping

OSI TP Service Parameters	CPI-C Calls, Parameters and Characteristics
if: Rollback = true	
TP-U-ABORT indication User-Data	*return_code* = CM_DEALLOCATED_ABEND_BO (S or R) *log_data* (characteristic)
if: Rollback = false	
TP-U-ABORT indication User-Data	*return_code* = CM_DEALLOCATED_ABEND (R) [+ CM_DEALLOCATED_ABEND (S)] *log_data* (characteristic)

Table D-39 TP-P-ABORT indication Mapping

OSI TP Service Parameters	CPI-C Calls, Parameters and Characteristics
if: Rollback = true	
TP-P-ABORT indication	Maps to *return_code* values:
Diagnostic	
permanent-failure	CM_RESOURCE_FAIL_NO_RETRY_BO (S or R)
transient-failure	CM_RESOURCE_FAILURE_RETRY_BO (S or R)
protocol-error	CM_RESOURCE_FAIL_NO_RETRY_BO (S or R)
begin-transaction-reject	CM_INCLUDE_PARTNER_REJECT_BO (S or R)
end-dialogue-collision	-
begin-transaction-end-dialogue-collision	CM_DEALLOCATED_ABEND_BO.dealloc_confirm_collision (S or R)
if: Rollback = false	
TP-P-ABORT indication	Maps to *return_code* values:
Diagnostic	
permanent-failure	CM_RESOURCE_FAILURE_NO_RETRY (R) [+ CM_RESOURCE_FAILURE_NO_RETRY (S)]
transient-failure	CM_RESOURCE_FAILURE_RETRY (R) [+ CM_RESOURCE_FAILURE_RETRY (S)]
protocol-error	CM_RESOURCE_FAILURE_NO_RETRY (R) [+ CM_RESOURCE_FAILURE_NO_RETRY (S)]
begin-transaction-reject	-
end-dialogue-collision	CM_DEALLOCATED_ABEND.dealloc_confirm_collision (R) [+ CM_DEALLOCATED_ABEND.dealloc_confirm_collision (S)]
begin-transaction-end-dialogue-collision	CM_DEALLOCATED_ABEND.begin_transaction_collision (R) [+CM_DEALLOCATED_ABEND.begin_transaction_collision (S)]

Table D-40 TP-DEFERRED-END-DIALOGUE and TP-PREPARE indication Mapping

OSI TP Service Parameters	CPI-C Calls, Parameters and Characteristics
if: a TP-DEFERRED-END-DIALOGUE indication has been received	
TP-PREPARE indication Data-Permitted	*status_received* = CM_TAKE_COMMIT_DEALLOCATE = false
if: a TP-DEFERRED-END-DIALOGUE indication has NOT been received	
TP-PREPARE indication Data-Permitted	*status_received* = CM_TAKE_COMMIT = false

D.2.2 Sequencing Rules and State Tables

The sequencing rules and state tables for the CPI-C ASE are contained in Appendix C.

D.2.3 CPI-C ASE Protocol Definition

There are no additional protocol definitions required by the CPI-C ASE beyond those defined in ISO/IEC 10026, Parts 1-6.

D.2.4 CPI-C ASE Structure and Encoding of APDUs

There are no additional APDUs required by the CPI-C ASE beyond those defined in ISO/IEC 10026, Parts 1-6.

D.3 LU 6.2 CRMs

This section 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 specification.

The CPI Communications calls have been built on top of the LU 6.2 verbs described in the referenced SNA Programmer's Reference specification. Table D-41 on page 483 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 Section D.3.1 on page 481).
- 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 Section D.3.1 on page 481).
- 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:

- 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.

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 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.

D.3.1 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 Section 4.3.6 on page 80.

D.3.2 Can CPI-C 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 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 Section D.3.3 for a discussion of naming conventions for service transaction programs.

D.3.3 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 on page 329 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'). See the referenced SNA Programmer's Reference specification 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.

D.3.4 Relationship between LU 6.2 Verbs and CPI Communications Calls

Table D-41 on page 483 shows CPI Communications calls on the left side and LU 6.2 verbs and their parameters across the top. The tables relate 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 column and a call row 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.

Note: The mapping for the following calls is applicable for half-duplex conversations only:

Confirm
Prepare_To_Receive
Request_To_Send
Set_Error_Direction
Set_Prepare_To_Receive_Type
Test_Request_To_Send_Received.

Table D-41 Relationship of LU 6.2 Verbs to CPI Communications Calls

CPI Communications Calls	**MC_ALLOCATE**	- LU_ NAME	- MODE_NAME	- TPN	- TYPE	- RETURN_CONTROL	- CONVERSATION_GROUP_ID	- SYNC_LEVEL	- SECURITY	- PIP	**MC_CONFIRM**	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED	**MC_CONFIRMED**	**MC_DEALLOCATE**	- TYPE	- EXPEDITED_DATA_RECEIVED	**MC_FLUSH**	**MC_GET_ATTRIBUTES**	- PARTNER_LU_NAME	- PART_FULL_QUAL_LU_NAME	- MODE_NAME	- SYNC_LEVEL	- CONVERSATION_STATE	- CONV_CORRELATOR	- SESSION_ID	- CONVERSATION_GROUP_ID
Starter Set																											
Accept_Conversation																D											
Allocate	X																										
Deallocate															X												
Initilize_Conversation	X	D	D	D	D	D		D	D	D						D											
Receive																											
Send_Data																											
Advanced Function																											
Accept_Incoming																											
Cancel_Conversation															X	D											
Confirm											X	X	X														
Confirmed														X													
Extract_Conversation_State																			X					X			
Extract_Conversation_Type																											
Extract_Mode_Name																			X			X					
Extract_Partner_LU_Name																			X	X	X						
Extract_Security_User_ID																											
Extract_Send_Receive_Mode																											
Extract_Sync_Level																			X				X				
Extract_TP_Name																											
Flush																	X										
Initialize_For_Incoming																D											
Prepare																											
Prepare_To_Receive																											
Receive_Expedited_Data																											
Request_To_Send																											
Send_Error																											
Send_Expedited_Data																											
Set_Confirmation_Urgency																											
Set_Conversation_Security_Password	X								S																		
Set_Conversation_Security_Type	X								S																		
Set_Conversation_Security_User_ID	X								S																		
Set_Conversation_Type	X																										
Set_Deallocate_Type															X	S											
Set_Error_Direction																											
Set_Fill																											
Set_Initialization_Data	X									S																	
Set_Log_Data																											
Set_Mode_Name	X		S																								
Set_Partner_LU_Name	X	S																									
Set_Prepare_To_Receive_Type																											
Set_Receive_Type																											
Set_Return_Control	X					S																					
Set_Send_Receive_Mode	X				S																						
Set_Send_Type																											
Set_Sync_Level	X							S																			
Set_TP_Name	X			S																							
Test_Request_To_Send_Received																											

CPI Communications Calls	MC_POST_ON_RECEIPT	MC_PREPARE_FOR_SYNCPT	MC_PREPARE_TO_RECEIVE	- TYPE	- LOCKS	MC_RECEIVE_AND_WAIT	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED	- WHAT_RECEIVED	- MAP_NAME	MC_RCV_EXPEDITED_DATA	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED	MC_RECEIVE_IMMEDIATE	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED	- WHAT_RECEIVED	- MAP_NAME	MC_REQUEST_TO_SEND	MC_SEND_DATA	- MAP_NAME	- USER_CONTROL_DATA	- ENCRYPT	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED
Starter Set																									
Accept_Conversation				D	D																				
Allocate																									
Deallocate																									
Initilize_Conversation				D	D																				
Receive						X	X	X	X					X	X	X	X								
Send_Data																				X				X	X
Advanced Function																									
Accept_Incoming																									
Cancel_Conversation																									
Confirm																									
Confirmed																									
Extract_Conversation_State																									
Extract_Conversation_Type																									
Extract_Mode_Name																									
Extract_Partner_LU_Name																									
Extract_Security_User_ID																									
Extract_Send_Receive_Mode																									
Extract_Sync_Level																									
Extract_TP_Name																									
Flush																									
Initialize_For_Incoming				D	D																				
Prepare		X																							
Prepare_To_Receive			X																						
Receive_Expedited_Data											X	X	X												
Request_To_Send																			X						
Send_Error																									
Send_Expedited_Data																									
Set_Confirmation_Urgency			X		S																				
Set_Conversation_Security_Password																									
Set_Conversation_Security_Type																									
Set_Conversation_Security_User_ID																									
Set_Conversation_Type																									
Set_Deallocate_Type																									
Set_Error_Direction																									
Set_Fill																									
Set_Initialization_Data																									
Set_Log_Data																									
Set_Mode_Name																									
Set_Partner_LU_Name																									
Set_Prepare_To_Receive_Type			X	S																					
Set_Receive_Type						X								X											
Set_Return_Control																									
Set_Send_Receive_Mode																									
Set_Send_Type																									
Set_Sync_Level																									
Set_TP_Name																									
Test_Request_To_Send_Received																									

CPI Communications Calls	**MC_SEND_ERROR**	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED	**MC_SEND_EXPEDITED_DATA**	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED	**MC_TEST**	- TEST_POSTED	- TEST_REQ_TO_SEND_RCVD	**BACKOUT**	**GET_TP_PROPERTIES**	- OWN_FULLY_QUAL_LU_NAME	- OWN_TP_NAME	- OWN_TP_INSTANCE	- SECURITY_USER_ID	- SECURITY_PROFILE	- LUW_IDENTIFIER	- PROTECTED_LUW_IDENTIFIER	**GET_TYPE**	**SET_SYNCPT_OPTIONS**	**SYNCPT**	- REQ_TO_SEND_RECEIVED	**WAIT**	- RESOURCE_POSTED	**WAIT_FOR_COMPLETION**
Starter Set																									
Accept_Conversation																									
Allocate																									
Deallocate																									
Initilize_Conversation																									
Receive																									
Send_Data																									
Advanced Function																									
Accept_Incoming																									
Cancel_Conversation																									
Confirm																									
Confirmed																									
Extract_Conversation_State																									
Extract_Conversation_Type																			X						
Extract_Mode_Name																									
Extract_Partner_LU_Name																									
Extract_Security_User_ID											X				X										
Extract_Send_Receive_Mode																			X						
Extract_Sync_Level																									
Extract_TP_Name											X		X												
Flush																									
Initialize_For_Incoming																									
Prepare																									
Prepare_To_Receive																									
Receive_Expedited_Data																									
Request_To_Send																									
Send_Error	X	X	X																						
Send_Expedited_Data				X	X	X																			
Set_Confirmation_Urgency																									
Set_Conversation_Security_Password																									
Set_Conversation_Security_Type																									
Set_Conversation_Security_User_ID																									
Set_Conversation_Type																									
Set_Deallocate_Type																									
Set_Error_Direction	X																								
Set_Fill																									
Set_Initialization_Data																									
Set_Log_Data																									
Set_Mode_Name																									
Set_Partner_LU_Name																									
Set_Prepare_To_Receive_Type																									
Set_Receive_Type																									
Set_Return_Control																									
Set_Send_Receive_Mode																									
Set_Send_Type																									
Set_Sync_Level																									
Set_TP_Name																									
Test_Request_To_Send_Received							X		X																

CPI Communications Calls	**ALLOCATE**	- LU_NAME	- MODE_NAME	- TPN	- TYPE	- RETURN_CONTROL	- CONVERSATION_GROUP_ID	- SYNC_LEVEL	- SECURITY	- PIP	**CONFIRM**	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED	**CONFIRMED**	**DEALLOCATE**	- TYPE	- LOG_DATA	- EXPEDITED_DATA_RECEIVED	**FLUSH**
Starter Set																			
Accept_Conversation																D	D		
Allocate	X																		
Deallocate															X				
Initilize_Conversation	X	D	D	D	D	D		D	D	D						D	D		
Receive																			
Send_Data																			
Advanced Function																			
Accept_Incoming																			
Cancel_Conversation																			
Confirm											X	X	X						
Confirmed														X					
Extract_Conversation_State																			
Extract_Conversation_Type																			
Extract_Mode_Name																			
Extract_Partner_LU_Name																			
Extract_Security_User_ID																			
Extract_Send_Receive_Mode																			
Extract_Sync_Level																			
Extract_TP_Name																			
Flush																			X
Initialize_For_Incoming																D	D		
Prepare																			
Prepare_To_Receive																			
Receive_Expedited_Data																			
Request_To_Send																			
Send_Error																			
Send_Expedited_Data																			
Set_Confirmation_Urgency																			
Set_Conversation_Security_Password	X								S										
Set_Conversation_Security_Type	X								S										
Set_Conversation_Security_User_ID	X								S										
Set_Conversation_Type	X				S														
Set_Deallocate_Type															X	S			
Set_Error_Direction																			
Set_Fill																			
Set_Initialization_Data	X									S									
Set_Log_Data															X		S		
Set_Mode_Name	X		S																
Set_Partner_LU_Name	X	S																	
Set_Prepare_To_Receive_Type																			
Set_Receive_Type																			
Set_Return_Control	X					S													
Set_Send_Receive_Mode	X				S														
Set_Send_Type																			
Set_Sync_Level	X							S											
Set_TP_Name	X			S															
Test_Request_To_Send_Received																			

CPI Communications Calls	**GET_ATTRIBUTES**	- PARTNER_LU_NAME	- PART_FULL_QUAL_LU_NAME	- MODE_NAME	- SYNC_LEVEL	- CONVERSATION_STATE	- CONV_CORRELATOR	- SESSION_ID	- CONVERSATION_GROUP_ID	**POST_ON_RECEIPT**	- FILL	**PREPARE_FOR_SYNCPT**	**PREAPARE_TO_RECEIVE**	- TYPE	- LOCKS	**RECEIVE_AND_WAIT**	- FILL	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED	- WHAT_RECEIVED	**RECEIVE_EXPEDITED_DATA**	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED
Starter Set																							
Accept_Conversation														D	D		D						
Allocate																							
Deallocate																							
Initilize_Conversation														D	D		D						
Receive																X		X	X	X			
Send_Data																							
Advanced Function																							
Accept_Incoming																							
Cancel_Conversation																							
Confirm																							
Confirmed																							
Extract_Conversation_State	X					X																	
Extract_Conversation_Type																							
Extract_Mode_Name	X			X																			
Extract_Partner_LU_Name	X	X	X																				
Extract_Security_User_ID																							
Extract_Send_Receive_Mode																							
Extract_Sync_Level	X				X																		
Extract_TP_Name																							
Flush																							
Initialize_For_Incoming														D	D		D						
Prepare												X											
Prepare_To_Receive													X										
Receive_Expedited_Data																					X	X	X
Request_To_Send																							
Send_Error																							
Send_Expedited_Data																							
Set_Confirmation_Urgency													X		S								
Set_Conversation_Security_Password																							
Set_Conversation_Security_Type																							
Set_Conversation_Security_User_ID																							
Set_Conversation_Type																							
Set_Deallocate_Type																							
Set_Error_Direction																							
Set_Fill																X	S						
Set_Initialization_Data																							
Set_Log_Data																							
Set_Mode_Name																							
Set_Partner_LU_Name																							
Set_Prepare_To_Receive_Type													X	S									
Set_Receive_Type																X							
Set_Return_Control																							
Set_Send_Receive_Mode																							
Set_Send_Type																							
Set_Sync_Level																							
Set_TP_Name																							
Test_Request_To_Send_Received																							

CPI Communications Calls	**RECEIVE_IMMEDIATE**	- FILL	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED	- WHAT_RECEIVED	**REQUEST_TO_SEND**	**SEND_DATA**	- ENCRYPT	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED	**SEND_EXPEDITED_DATA**	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED	**SEND_ERROR**	- TYPE	- LOG_DATA	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED	**TEST**	- TEST_POSTED	- TEST_REQ_TO_SEND_RCVD
Starter Set																					
Accept_Conversation		D														D					
Allocate																					
Deallocate																					
Initilize_Conversation		D														D					
Receive	X		X	X	X																
Send_Data							X		X	X											
Advanced Function																					
Accept_Incoming																					
Cancel_Conversation																					
Confirm																					
Confirmed																					
Extract_Conversation_State																					
Extract_Conversation_Type																					
Extract_Mode_Name																					
Extract_Partner_LU_Name																					
Extract_Security_User_ID																					
Extract_Send_Receive_Mode																					
Extract_Sync_Level																					
Extract_TP_Name																					
Flush																					
Initialize_For_Incoming		D															D				
Prepare																					
Prepare_To_Receive																					
Receive_Expedited_Data																					
Request_To_Send						X															
Send_Error														X	D		X	X			
Send_Expedited_Data											X	X	X								
Set_Confirmation_Urgency																					
Set_Conversation_Security_Password																					
Set_Conversation_Security_Type																					
Set_Conversation_Security_User_ID																					
Set_Conversation_Type																					
Set_Deallocate_Type																					
Set_Error_Direction														X							
Set_Fill	X	S																			
Set_Initialization_Data																					
Set_Log_Data														X		S					
Set_Mode_Name																					
Set_Partner_LU_Name																					
Set_Prepare_To_Receive_Type																					
Set_Receive_Type	X																				
Set_Return_Control																					
Set_Send_Receive_Mode																					
Set_Send_Type																					
Set_Sync_Level																					
Set_TP_Name																					
Test_Request_To_Send_Received																			X		X

Appendix E

Pseudonym Files

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

E.1 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.

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                                                                       *
 *         CPI Communications Pseudonyms -- SC31-6180-01                 *
 *                                                                       *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * 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


#define CM_INT32 signed long int


#define CM_ENTRY extern void
#define CM_PTR *


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

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

typedef CM_INT32 CM_AE_QUAL_OR_AP_TITLE_FORMAT;
                                    /* used for AE_QUALIFIER_FORMAT  */
                                    /* and AP_TITLE_FORMAT           */
typedef CM_INT32 CM_ALLOCATE_CONFIRM_TYPE;
typedef CM_INT32 CM_BEGIN_TRANSACTION;
typedef CM_INT32 CM_BUFFER_LENGTH;
typedef CM_INT32 CM_CALL_ID;
typedef CM_INT32 CM_COMPLETED_OP_COUNT;
typedef CM_INT32 CM_CONFIRMATION_URGENCY;
typedef CM_INT32 CM_CONTROL_INFORMATION_RECEIVED;
```

```
typedef CM_INT32 CM_CONVERSATION_QUEUE;
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_DIRECTORY_ENCODING;
typedef CM_INT32 CM_DIRECTORY_SYNTAX;
typedef CM_INT32 CM_ERROR_DIRECTION;
typedef CM_INT32 CM_FILL;
typedef CM_INT32 CM_JOIN_TRANSACTION_TYPE;
typedef CM_INT32 CM_MAXIMUM_BUFFER_SIZE;
typedef CM_INT32 CM_OOID;
typedef CM_INT32 CM_OOID_LIST_COUNT;
typedef CM_INT32 CM_PREPARE_DATA_PERMITTED_TYPE;
typedef CM_INT32 CM_PREPARE_TO_RECEIVE_TYPE;
typedef CM_INT32 CM_PROCESSING_MODE;  /* also used for queue_processing_mode  */
typedef CM_INT32 CM_RECEIVE_TYPE;     /* also used for expedited_receive_type */
typedef CM_CONTROL_INFORMATION_RECEIVED  CM_REQUEST_TO_SEND_RECEIVED;
typedef CM_INT32 CM_RETURN_CODE;
typedef CM_INT32 CM_RETURN_CONTROL;
typedef CM_INT32 CM_SEND_RECEIVE_MODE;
typedef CM_INT32 CM_SEND_TYPE;
typedef CM_INT32 CM_STATUS_RECEIVED;
typedef CM_INT32 CM_SYNC_LEVEL;
typedef CM_INT32 CM_TIMEOUT;
typedef CM_INT32 CM_TRANSACTION_CONTROL;

/* X/open typedefs for compatibility  */
typedef CM_INT32 CONVERSATION_TYPE;
typedef CM_INT32 CONVERSATION_SECURITY_TYPE;
typedef CM_INT32 DATA_RECEIVED;
typedef CM_INT32 DEALLOCATE_TYPE;
typedef CM_INT32 ERROR_DIRECTION;
typedef CM_INT32 PREPARE_TO_RECEIVE_TYPE;
typedef CM_INT32 PROCESSING_MODE;
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;

/*
 * 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.
 */

/*  AE_qual_or_AP_title_format  values, used for
    AE_qualifier_format  and AP_title_format  parameters */

#define CM_DN                           (CM_AE_QUAL_OR_AP_TITLE_FORMAT)  0
#define CM_OID                          (CM_AE_QUAL_OR_AP_TITLE_FORMAT)  1
```

```
#define CM_INT_DIGITS                   (CM_AE_QUAL_OR_AP_TITLE_FORMAT)  2


/*  allocate_confirm values  */

#define CM_ALLOCATE_NO_CONFIRM          (CM_ALLOCATE_CONFIRM_TYPE)  0
#define CM_ALLOCATE_CONFIRM             (CM_ALLOCATE_CONFIRM_TYPE)  1


/*  begin_transaction values  */

#define CM_BEGIN_IMPLICIT               (CM_BEGIN_TRANSACTION)  0
#define CM_BEGIN_EXPLICIT               (CM_BEGIN_TRANSACTION)  1


/*  call_ID values  */

#define CM_CMACCI                       (CM_CALL_ID) 1
#define CM_CMACCP                       (CM_CALL_ID) 2
#define CM_CMALLC                       (CM_CALL_ID) 3
#define CM_CMCANC                       (CM_CALL_ID) 4
#define CM_CMCFM                        (CM_CALL_ID) 5
#define CM_CMCFMD                       (CM_CALL_ID) 6
#define CM_CMCNVI                       (CM_CALL_ID) 7
#define CM_CMCNVO                       (CM_CALL_ID) 8
#define CM_CMDEAL                       (CM_CALL_ID) 9
#define CM_CMDFDE                       (CM_CALL_ID) 10
#define CM_CMEACN                       (CM_CALL_ID) 11
#define CM_CMEAEQ                       (CM_CALL_ID) 12
#define CM_CMEAPT                       (CM_CALL_ID) 13
#define CM_CMECS                        (CM_CALL_ID) 14
#define CM_CMECT                        (CM_CALL_ID) 15
#define CM_CMECTX                       (CM_CALL_ID) 16
#define CM_CMEID                        (CM_CALL_ID) 17
#define CM_CMEMBS                       (CM_CALL_ID) 18
#define CM_CMEMN                        (CM_CALL_ID) 19
#define CM_CMEPLN                       (CM_CALL_ID) 21
#define CM_CMESI                        (CM_CALL_ID) 22
#define CM_CMESL                        (CM_CALL_ID) 23
#define CM_CMESRM                       (CM_CALL_ID) 24
#define CM_CMESUI                       (CM_CALL_ID) 25
#define CM_CMETC                        (CM_CALL_ID) 26
#define CM_CMETPN                       (CM_CALL_ID) 27
#define CM_CMFLUS                       (CM_CALL_ID) 28
#define CM_CMINCL                       (CM_CALL_ID) 29
#define CM_CMINIC                       (CM_CALL_ID) 30
#define CM_CMINIT                       (CM_CALL_ID) 31
#define CM_CMPREP                       (CM_CALL_ID) 32
#define CM_CMPTR                        (CM_CALL_ID) 33
#define CM_CMRCV                        (CM_CALL_ID) 34
#define CM_CMRCVX                       (CM_CALL_ID) 35
#define CM_CMRLTP                       (CM_CALL_ID) 36
#define CM_CMRTS                        (CM_CALL_ID) 37
#define CM_CMSAC                        (CM_CALL_ID) 38
#define CM_CMSACN                       (CM_CALL_ID) 39
#define CM_CMSAEQ                       (CM_CALL_ID) 40
#define CM_CMSAPT                       (CM_CALL_ID) 41
#define CM_CMSBT                        (CM_CALL_ID) 42
#define CM_CMSCSP                       (CM_CALL_ID) 43
```

```
#define CM_CMSCST                       (CM_CALL_ID) 44
#define CM_CMSCSU                       (CM_CALL_ID) 45
#define CM_CMSCT                        (CM_CALL_ID) 46
#define CM_CMSCU                        (CM_CALL_ID) 47
#define CM_CMSDT                        (CM_CALL_ID) 48
#define CM_CMSED                        (CM_CALL_ID) 49
#define CM_CMSEND                       (CM_CALL_ID) 50
#define CM_CMSERR                       (CM_CALL_ID) 51
#define CM_CMSF                         (CM_CALL_ID) 52
#define CM_CMSID                        (CM_CALL_ID) 53
#define CM_CMSLD                        (CM_CALL_ID) 54
#define CM_CMSLTP                       (CM_CALL_ID) 55
#define CM_CMSMN                        (CM_CALL_ID) 56
#define CM_CMSNDX                       (CM_CALL_ID) 57
#define CM_CMSPDP                       (CM_CALL_ID) 58
#define CM_CMSPLN                       (CM_CALL_ID) 60
#define CM_CMSPM                        (CM_CALL_ID) 61
#define CM_CMSPTR                       (CM_CALL_ID) 62
#define CM_CMSQCF                       (CM_CALL_ID) 63
#define CM_CMSQPM                       (CM_CALL_ID) 64
#define CM_CMSRC                        (CM_CALL_ID) 65
#define CM_CMSRT                        (CM_CALL_ID) 66
#define CM_CMSSL                        (CM_CALL_ID) 67
#define CM_CMSSRM                       (CM_CALL_ID) 68
#define CM_CMSST                        (CM_CALL_ID) 69
#define CM_CMSTC                        (CM_CALL_ID) 70
#define CM_CMSTPN                       (CM_CALL_ID) 71
#define CM_CMTRTS                       (CM_CALL_ID) 72
#define CM_CMWAIT                       (CM_CALL_ID) 73
#define CM_CMWCMP                       (CM_CALL_ID) 74
#define CM_CMSJT                        (CM_CALL_ID) 75


/*  confirmation_urgency  values  */

#define CM_CONFIRMATION_NOT_URGENT      (CM_CONFIRMATION_URGENCY)  0
#define CM_CONFIRMATION_URGENT          (CM_CONFIRMATION_URGENCY)  1


/*  control_information_received,  request_to_send_received  values  */

#define CM_NO_CONTROL_INFO_RECEIVED     (CM_CONTROL_INFORMATION_RECEIVED)   0
#define CM_REQ_TO_SEND_NOT_RECEIVED     (CM_CONTROL_INFORMATION_RECEIVED)   0
#define CM_REQ_TO_SEND_RECEIVED         (CM_CONTROL_INFORMATION_RECEIVED)   1
#define CM_ALLOCATE_CONFIRMED           (CM_CONTROL_INFORMATION_RECEIVED)   2
#define CM_ALLOCATE_CONFIRMED_WITH_DATA (CM_CONTROL_INFORMATION_RECEIVED)   3
#define CM_ALLOCATE_REJECTED_WITH_DATA  (CM_CONTROL_INFORMATION_RECEIVED)   4
#define CM_EXPEDITED_DATA_AVAILABLE     (CM_CONTROL_INFORMATION_RECEIVED)   5
#define CM_RTS_RCVD_AND_EXP_DATA_AVAIL  (CM_CONTROL_INFORMATION_RECEIVED)   6


/*  conversation_queue  values  */

#define CM_INITIALIZATION_QUEUE         (CM_CONVERSATION_QUEUE)  0
#define CM_SEND_QUEUE                   (CM_CONVERSATION_QUEUE)  1
#define CM_RECEIVE_QUEUE                (CM_CONVERSATION_QUEUE)  2
#define CM_SEND_RECEIVE_QUEUE           (CM_CONVERSATION_QUEUE)  3
#define CM_EXPEDITED_SEND_QUEUE         (CM_CONVERSATION_QUEUE)  4
#define CM_EXPEDITED_RECEIVE_QUEUE      (CM_CONVERSATION_QUEUE)  5
```

```
/*  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
#define CM_SEND_ONLY_STATE               (CM_CONVERSATION_STATE)  15
#define CM_RECEIVE_ONLY_STATE            (CM_CONVERSATION_STATE)  16
#define CM_SEND_RECEIVE_STATE            (CM_CONVERSATION_STATE)  17
#define CM_PREPARED_STATE                (CM_CONVERSATION_STATE)  18


/*  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


/* join transaction values */

#define CM_JOIN_IMPLICIT                 (CM_JOIN_TRANSACTION_TYPE)  0
#define CM_JOIN_EXPLICIT                 (CM_JOIN_TRANSACTION_TYPE)  1
```

```
/*  prepare_data_permitted  values  */

#define CM_PREPARE_DATA_NOT_PERMITTED    (CM_PREPARE_DATA_PERMITTED_TYPE)  0
#define CM_PREPARE_DATA_PERMITTED        (CM_PREPARE_DATA_PERMITTED_TYPE)  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


/*  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_SYS    (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_CONVERSATION_ENDING           (CM_RETURN_CODE) 38   /* CPIC 2.0 */
#define CM_SEND_RCV_MODE_NOT_SUPPORTED   (CM_RETURN_CODE) 39   /* CPIC 2.0 */
#define CM_BUFFER_TOO_SMALL              (CM_RETURN_CODE) 40   /* CPIC 2.0 */
#define CM_EXP_DATA_NOT_SUPPORTED        (CM_RETURN_CODE) 41   /* CPIC 2.0 */
#define CM_DEALLOC_CONFIRM_REJECT        (CM_RETURN_CODE) 42   /* CPIC 2.0 */
#define CM_ALLOCATION_ERROR              (CM_RETURN_CODE) 43   /* CPIC 2.0 */
#define CM_RETRY_LIMIT_EXCEEDED          (CM_RETURN_CODE) 44   /* CPIC 2.0 */
#define CM_NO_SECONDARY_INFORMATION      (CM_RETURN_CODE) 45   /* CPIC 2.0 */
#define CM_SECURITY_NOT_SUPPORTED        (CM_RETURN_CODE) 46   /* CPIC 2.0 */
#define CM_CALL_NOT_SUPPORTED            (CM_RETURN_CODE) 48   /* CPIC 2.0 */
#define CM_PARM_VALUE_NOT_SUPPORTED      (CM_RETURN_CODE) 49   /* CPIC 2.0 */
#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
#define CM_CONV_DEALLOC_AFTER_SYNCPT     (CM_RETURN_CODE) 136  /* CPIC 2.0 */
#define CM_INCLUDE_PARTNER_REJECT_BO     (CM_RETURN_CODE) 137  /* CPIC 2.0 */


/*  return_control values  */

#define CM_WHEN_SESSION_ALLOCATED        (CM_RETURN_CONTROL) 0
#define CM_IMMEDIATE                     (CM_RETURN_CONTROL) 1


/*  send_receive_mode values  */

#define CM_HALF_DUPLEX                   (CM_SEND_RECEIVE_MODE) 0
#define CM_FULL_DUPLEX                   (CM_SEND_RECEIVE_MODE) 1


/*  send_type values  */

#define CM_BUFFER_DATA                   (CM_SEND_TYPE) 0
#define CM_SEND_AND_FLUSH                (CM_SEND_TYPE) 1
#define CM_SEND_AND_CONFIRM              (CM_SEND_TYPE) 2
#define CM_SEND_AND_PREP_TO_RECEIVE      (CM_SEND_TYPE) 3
#define CM_SEND_AND_DEALLOCATE           (CM_SEND_TYPE) 4


/*  status_received values  */

#define CM_NO_STATUS_RECEIVED            (CM_STATUS_RECEIVED) 0
#define CM_SEND_RECEIVED                 (CM_STATUS_RECEIVED) 1
#define CM_CONFIRM_RECEIVED              (CM_STATUS_RECEIVED) 2
#define CM_CONFIRM_SEND_RECEIVED         (CM_STATUS_RECEIVED) 3
#define CM_CONFIRM_DEALLOC_RECEIVED      (CM_STATUS_RECEIVED) 4
#define CM_TAKE_COMMIT                   (CM_STATUS_RECEIVED) 5
#define CM_TAKE_COMMIT_SEND              (CM_STATUS_RECEIVED) 6
#define CM_TAKE_COMMIT_DEALLOCATE        (CM_STATUS_RECEIVED) 7
#define CM_TAKE_COMMIT_DATA_OK           (CM_STATUS_RECEIVED) 8
#define CM_TAKE_COMMIT_SEND_DATA_OK      (CM_STATUS_RECEIVED) 9
#define CM_TAKE_COMMIT_DEALLOC_DATA_OK   (CM_STATUS_RECEIVED) 10
#define CM_PREPARE_OK                    (CM_STATUS_RECEIVED) 11
#define CM_JOIN_TRANSACTION              (CM_STATUS_RECEIVED) 12
```

```
/* 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
#define CM_SYNC_POINT_NO_CONFIRM         (CM_SYNC_LEVEL) 3

/* 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
#define CM_SECURITY_PROGRAM_STRONG       (CM_CONVERSATION_SECURITY_TYPE)  5

/* transaction_control  values */

#define CM_CHAINED_TRANSACTIONS          (CM_TRANSACTION_CONTROL)  0
#define CM_UNCHAINED_TRANSACTIONS        (CM_TRANSACTION_CONTROL)  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_CONTROL_INFORMATION_RECEIVED  CM_PTR,
                                               /* control_information_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              */
CM_ENTRY cmdfde(unsigned  char CM_PTR,         /* conversation_ID          */
                CM_RETURN_CODE  CM_PTR);       /* return_code              */
CM_ENTRY cmeaeq(unsigned  char CM_PTR,         /* conversation_ID          */
                unsigned char CM_PTR,          /* AE_qualifier             */
                CM_INT32 CM_PTR,               /* AE_qualifier_length      */
                CM_AE_QUAL_OR_AP_TITLE_FORMAT  CM_PTR,
                                               /* AE_qualifier_format      */
                CM_RETURN_CODE  CM_PTR);       /* return_code              */
CM_ENTRY cmeapt(unsigned  char CM_PTR,         /* conversation_ID          */
```

```
                unsigned char CM_PTR,          /* AP_title                */
                CM_INT32 CM_PTR,               /* AP_title length         */
                CM_AE_QUAL_OR_AP_TITLE_FORMAT  CM_PTR,
                                               /* AP_title format         */
                CM_RETURN_CODE CM_PTR);        /* return_code             */
CM_ENTRY cmeacn(unsigned char CM_PTR,          /* conversation_ID         */
                unsigned char CM_PTR,          /* application_context_name */
                CM_INT32 CM_PTR,               /* appl_context_name_length */
                CM_RETURN_CODE CM_PTR);        /* return_code             */
CM_ENTRY cmecs( unsigned char CM_PTR,          /* conversation_ID         */
                CM_CONVERSATION_STATE CM_PTR,  /* conversation_state      */
                CM_RETURN_CODE CM_PTR);        /* return_code             */
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 cmeid( unsigned char CM_PTR,          /* conversation_ID         */
                unsigned char CM_PTR,          /* initialization_data     */
                CM_INT32 CM_PTR,               /* requested_length        */
                CM_INT32 CM_PTR,               /* initialization_data_leng */
                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 cmesi( unsigned char CM_PTR,          /* conversation_ID         */
                CM_INT32 CM_PTR,               /* call_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_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 cmesrm(unsigned char CM_PTR,          /* conversation_ID         */
                CM_SEND_RECEIVE_MODE CM_PTR,   /* send_receive_mode       */
                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 cmetc( unsigned char CM_PTR,          /* conversation_ID         */
                CM_TRANSACTION_CONTROL CM_PTR,/* transaction_control      */
                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 cmincl(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 cmprep(unsigned char CM_PTR,         /* conversation_ID          */
                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_CONTROL_INFORMATION_RECEIVED  CM_PTR,
                                              /* control_information_received  */
                CM_RETURN_CODE CM_PTR);       /* return_code              */
CM_ENTRY cmrcvx(unsigned char CM_PTR,         /* conversation_ID          */
                unsigned char CM_PTR,         /* buffer                   */
                CM_INT32 CM_PTR,              /* requested_length         */
                CM_INT32 CM_PTR,              /* received_length          */
                CM_CONTROL_INFORMATION_RECEIVED  CM_PTR,
                                              /* control_information_received  */
                CM_RECEIVE_TYPE  CM_PTR,      /* expedited_receive_type   */
                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 cmsaeq(unsigned char  CM_PTR,        /* conversation_ID          */
                unsigned char  CM_PTR,        /* AE_qualifier             */
                CM_INT32 CM_PTR,              /* AE_qualifier  length     */
                CM_AE_QUAL_OR_AP_TITLE_FORMAT  CM_PTR,
                                              /* AE_qualifier  format     */
                CM_RETURN_CODE CM_PTR);       /* return_code              */
CM_ENTRY cmsac( unsigned char CM_PTR,         /* conversation_ID          */
                CM_ALLOCATE_CONFIRM_TYPE  CM_PTR,
                                              /* allocate_confirm         */
                CM_RETURN_CODE CM_PTR);       /* return_code              */
CM_ENTRY cmsacn(unsigned char  CM_PTR,        /* conversation_ID          */
                unsigned char  CM_PTR,        /* application_context_name */
                CM_INT32 CM_PTR,              /* appl_context_name_length */
                CM_RETURN_CODE CM_PTR);       /* return_code              */
CM_ENTRY cmsapt(unsigned char  CM_PTR,        /* conversation_ID          */
                unsigned char  CM_PTR,        /* AP_title                 */
                CM_INT32 CM_PTR,              /* AP_title_length          */
                CM_AE_QUAL_OR_AP_TITLE_FORMAT  CM_PTR,
                                              /* AP_title_format          */
                CM_RETURN_CODE CM_PTR);       /* return_code              */
CM_ENTRY cmsbt( unsigned char CM_PTR,         /* conversation_ID          */
                CM_BEGIN_TRANSACTION  CM_PTR,  /* begin_transaction       */
                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_CONVERSATION_SECURITY_TYPE  CM_PTR,
```

```
                                          /* conv_security_type       */
                CM_RETURN_CODE CM_PTR);   /* return_code              */
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 cmscu( unsigned char CM_PTR,     /* conversation_ID          */
                CM_CONFIRMATION_URGENCY CM_PTR,
                                          /* confirmation_urgency     */
                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_CONTROL_INFORMATION_RECEIVED CM_PTR,
                                          /* control_information_received */
                CM_RETURN_CODE CM_PTR);   /* return_code              */
CM_ENTRY cmserr(unsigned char CM_PTR,     /* conversation_ID          */
                CM_CONTROL_INFORMATION_RECEIVED CM_PTR,
                                          /* control_information_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 cmsid( unsigned char CM_PTR,     /* conversation_ID          */
                unsigned char CM_PTR,     /* initialization_data      */
                CM_INT32 CM_PTR,          /* init_data length         */
                CM_RETURN_CODE CM_PTR);   /* return_code              */
CM_ENTRY cmsjt( unsigned char CM_PTR,     /* conversation_ID          */
                CM_JOIN_TRANSACTION_TYPE CM_PTR,
                                          /* join_transaction         */
                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 cmsndx(unsigned char CM_PTR,     /* conversation_ID          */
                unsigned char CM_PTR,     /* buffer                   */
                CM_INT32 CM_PTR,          /* send_length              */
                CM_CONTROL_INFORMATION_RECEIVED CM_PTR,
                                          /* control_information_received */
                CM_RETURN_CODE CM_PTR);   /* return_code              */
CM_ENTRY cmspdp(unsigned char CM_PTR,     /* conversation_ID          */
                CM_PREPARE_DATA_PERMITTED_TYPE CM_PTR,
```

```
                                          /* prepare_data_permitted   */
                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 cmsqcf(unsigned char CM_PTR,     /* conversation_ID          */
                CM_CONVERSATION_QUEUE CM_PTR, /* conversation_queue   */
                void CM_PTR,              /* callback_function        */
                unsigned char CM_PTR,     /* user_field               */
                CM_RETURN_CODE CM_PTR);   /* return_code              */
CM_ENTRY cmsqpm(unsigned char CM_PTR,     /* conversation_ID          */
                CM_CONVERSATION_QUEUE CM_PTR, /* conversation_queue   */
                CM_PROCESSING_MODE CM_PTR,/* queue_processing_mode    */
                unsigned char CM_PTR,     /* user_field               */
                CM_INT32 CM_PTR,          /* OOID                     */
                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_SYNC_LEVEL CM_PTR,     /* sync_level               */
                CM_RETURN_CODE CM_PTR);   /* return_code              */
CM_ENTRY cmssrm(unsigned char CM_PTR,     /* conversation_ID          */
                CM_SEND_RECEIVE_MODE CM_PTR, /* send_receive_mode     */
                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 cmstc( unsigned char CM_PTR,     /* conversation_ID          */
                CM_TRANSACTION_CONTROL CM_PTR,/* transaction_control  */
                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_CONTROL_INFORMATION_RECEIVED CM_PTR,
                                          /* control_information_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              */
CM_ENTRY cmwcmp(unsigned char CM_PTR,     /* OOID_list                */
                CM_OOID_LIST_COUNT CM_PTR,/* OOID_list_count          */
                CM_INT32 CM_PTR,          /* timeout                  */
                unsigned char CM_PTR,     /* completed_op_index_list  */
                CM_INT32 CM_PTR,          /* completed_op_count       */
                unsigned char CM_PTR,     /* user_field_list          */
```

```
                CM_RETURN_CODE  CM_PTR);       /* return_code              */


#ifdef __cplusplus
}
#endif /* __cplusplus */


/*
 * 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 Deferred_Deallocate(v1,v2)             cmdfde(v1,v2)
#define Extract_AE_Qualifier(v1,v2,v3,v4,v5)   cmeaeq(v1,v2,v3,v4,v5)
#define Extract_AP_Title(v1,v2,v3,v4,v5)       cmeapt(v1,v2,v3,v4,v5)
#define Extract_Application_Context_Name(v1,v2,v3,v4)  cmeacn(v1,v2,v3,v4)
#define Extract_Conversation_State(v1,v2,v3)   cmecs(v1,v2,v3)
#define Extract_Conversation_Type(v1,v2,v3)    cmect(v1,v2,v3)
#define Extract_Initialization_Data(v1,v2,v3,v4,v5)  cmeid(v1,v2,v3,v4,v5)
#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_Secondary_Information(v1,v2,v3,v4,v5,v6,v7)  \
                                               cmesi(v1,v2,v3,v4,v5,v6,v7)
#define Extract_Security_User_ID(v1,v2,v3,v4)  cmesui(v1,v2,v3,v4)
#define Extract_Send_Receive_Mode(v1,v2,v3)    cmesrm(v1,v2,v3)
#define Extract_Sync_Level(v1,v2,v3)           cmesl(v1,v2,v3)
#define Extract_Transaction_Control(v1,v2,v3)  cmetc(v1,v2,v3)
#define Extract_TP_Name(v1,v2,v3,v4)           cmetpn(v1,v2,v3,v4)
#define Flush(v1,v2)                           cmflus(v1,v2)
#define Include_Partner_In_Transaction(v1,v2)  cmincl(v1,v2)
#define Initialize_Conversation(v1,v2,v3)      cminit(v1,v2,v3)
#define Initialize_For_Incoming(v1,v2)         cminic(v1,v2)
#define Prepare(v1,v2)                         cmprep(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 Receive_Expedited_Data(v1,v2,v3,v4,v5,v6,v7)  cmrcvx(v1,v2,v3,v4,v5,v6,v7)
#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 Send_Expedited_Data(v1,v2,v3,v4,v5)    cmsndx(v1,v2,v3,v4,v5)
#define Set_AE_Qualifier(v1,v2,v3,v4,v5)       cmsaeq(v1,v2,v3,v4,v5)
#define Set_Allocate_Confirm(v1,v2,v3)         cmsac(v1,v2,v3)
```

```
#define Set_AP_Title(v1,v2,v3,v4,v5)           cmsapt(v1,v2,v3,v4,v5)
#define Set_Application_Context_Name(v1,v2,v3,v4)  cmsacn(v1,v2,v3,v4)
#define Set_Begin_Transaction(v1,v2,v3)        cmsbt(v1,v2,v3)
#define Set_Confirmation_Urgency(v1,v2,v3)     cmscu(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_Initialization_Data(v1,v2,v3,v4)   cmsid(v1,v2,v3,v4)
#define Set_Join_Transaction(v1,v2,v3)         cmsjt(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_Data_Permitted(v1,v2,v3)   cmspdp(v1,v2,v3)
#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_Queue_Callback_Function(v1,v2,v3,v4,v5)  cmsqcf(v1,v2,v3,v4,v5)
#define Set_Queue_Processing_Mode(v1,v2,v3,v4,v5,v6)  cmsqpm(v1,v2,v3,v4,v5,v6)
#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_Receive_Mode(v1,v2,v3)        cmssrm(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 Set_Transaction_Control(v1,v2,v3)      cmstc(v1,v2,v3)
#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_Completion(v1,v2,v3,v4,v5,v6,v7)  cmwcmp(v1,v2,v3,v4,v5,v6,v7)
#define Wait_For_Conversation(v1,v2,v3)        cmwait(v1,v2,v3)

#endif

/*
 *  The following list provides compatibility with the former
 *  version of X/Open CPI-C published as an X/Open CAE specification
 *  in February 1992.
 */
#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 CMDFDE cmdfde
#define CMEACN cmeacn
#define CMEAEQ cmeaeq
#define CMEAPT cmeapt
#define CMECS  cmecs
#define CMECT  cmect
#define CMEID  cmeid
#define CMEMBS cmembs
#define CMEMN  cmemn
#define CMEPLN cmepln
```

```
#define CMESI  cmesi
#define CMESL  cmesl
#define CMESRM cmesrm
#define CMESUI cmesui
#define CMETC  cmetc
#define CMETPN cmetpn
#define CMFLUS cmflus
#define CMINCL cmincl
#define CMINIC cminic
#define CMINIT cminit
#define CMPREP cmprep
#define CMPTR  cmptr
#define CMRCV  cmrcv
#define CMRCVX cmrcvx
#define CMRLTP cmrltp
#define CMRTS  cmrts
#define CMSAC  cmsac
#define CMSACN cmsacn
#define CMSAEQ cmsaeq
#define CMSAPT cmsapt
#define CMSBT  cmsbt
#define CMSCSP cmscsp
#define CMSCST cmscst
#define CMSCSU cmscsu
#define CMSCT  cmsct
#define CMSCU  cmscu
#define CMSDT  cmsdt
#define CMSED  cmsed
#define CMSEND cmsend
#define CMSERR cmserr
#define CMSF   cmsf
#define CMSID  cmsid
#define CMSJT  cmsjt
#define CMSLD  cmsld
#define CMSLTP cmsltp
#define CMSMN  cmsmn
#define CMSNDX cmsndx
#define CMSPDP cmspdp
#define CMSPLN cmspln
#define CMSPM  cmspm
#define CMSPTR cmsptr
#define CMSQCF cmsqcf
#define CMSQPM cmsqpm
#define CMSRC  cmsrc
#define CMSRT  cmsrt
#define CMSSL  cmssl
#define CMSSRM cmssrm
#define CMSST  cmsst
#define CMSTC  cmstc
#define CMSTPN cmstpn
#define CMTRTS cmtrts
#define CMWAIT cmwait
#define CMWCMP cmwcmp

#endif

/* ********************** End of Pseudonyms ********************** */
```

E.2 COBOL Pseudonym File (CMCOBOL)

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                   *
*       CPI COMMUNICATIONS PSEUDONYMS -- SC31-6180-01               *
*                                                                   *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* NOTE: BUFFER MUST BE DEFINED IN WORKING STORAGE
*
*
 01  AE-QUALIFIER                 PIC X(1024).
*    0-1024 BYTES
*
*  AE-QUALIFIER-FORMAT  USES AE-QUAL-OR-AP-TITLE-FORMAT  VALUES
*
 01  AE-QUAL-OR-AP-TITLE-FORMAT   PIC 9(9) COMP-5.
     88  CM-DN                             VALUE 0.
     88  CM-OID                            VALUE 1.
     88  CM-INT-DIGITS                     VALUE 2.
*
 01  AE-QUALIFIER-LENGTH          PIC 9(9) COMP-5.
*
 01  ALLOCATE-CONFIRM             PIC 9(9) COMP-5.
     88  CM-ALLOCATE-NO-CONFIRM            VALUE 0.
     88  CM-ALLOCATE-CONFIRM               VALUE 1.
*
 01  APPLICATION-CONTEXT-NAME     PIC X(256).
*    0-256 BYTES
*
 01  APPLICATION-CONTEXT-NAME-LEN  PIC 9(9) COMP-5.
*
 01  AP-TITLE                     PIC X(1024).
*    0-1024 BYTES
*
*  AP-TITLE-FORMAT  USES AE-QUAL-OR-AP-TITLE-FORMAT  VALUES
*
 01  AP-TITLE-LENGTH              PIC 9(9) COMP-5.
*
 01  BEGIN-TRANSACTION            PIC 9(9) COMP-5.
     88  CM-BEGIN-IMPLICIT                 VALUE 0.
     88  CM-BEGIN-EXPLICIT                 VALUE 1.
*
 01  BUFFER-LENGTH                PIC 9(9) COMP-5.
*
 01  CALL-ID                      PIC 9(9) COMP-5.
     88   CM-CMACCI                        VALUE 1.
     88   CM-CMACCP                        VALUE 2.
     88   CM-CMALLC                        VALUE 3.
     88   CM-CMCANC                        VALUE 4.
     88   CM-CMCFM                         VALUE 5.
     88   CM-CMCFMD                        VALUE 6.
     88   CM-CMCNVI                        VALUE 7.
     88   CM-CMCNVO                        VALUE 8.
     88   CM-CMDEAL                        VALUE 9.
     88   CM-CMDFDE                        VALUE 10.
     88   CM-CMEACN                        VALUE 11.
     88   CM-CMEAEQ                        VALUE 12.
     88   CM-CMEAPT                        VALUE 13.
```

```
88   CM-CMECS                    VALUE 14.
88   CM-CMECT                    VALUE 15.
88   CM-CMEID                    VALUE 17.
88   CM-CMEMBS                   VALUE 18.
88   CM-CMEMN                    VALUE 19.
88   CM-CMEPLN                   VALUE 21.
88   CM-CMESI                    VALUE 22.
88   CM-CMESL                    VALUE 23.
88   CM-CMESRM                   VALUE 24.
88   CM-CMESUI                   VALUE 25.
88   CM-CMETC                    VALUE 26.
88   CM-CMETPN                   VALUE 27.
88   CM-CMFLUS                   VALUE 28.
88   CM-CMINCL                   VALUE 29.
88   CM-CMINIC                   VALUE 30.
88   CM-CMINIT                   VALUE 31.
88   CM-CMPREP                   VALUE 32.
88   CM-CMPTR                    VALUE 33.
88   CM-CMRCV                    VALUE 34.
88   CM-CMRCVX                   VALUE 35.
88   CM-CMRLTP                   VALUE 36.
88   CM-CMRTS                    VALUE 37.
88   CM-CMSAC                    VALUE 38.
88   CM-CMSACN                   VALUE 39.
88   CM-CMSAEQ                   VALUE 40.
88   CM-CMSAPT                   VALUE 41.
88   CM-CMSBT                    VALUE 42.
88   CM-CMSCSP                   VALUE 43.
88   CM-CMSCST                   VALUE 44.
88   CM-CMSCSU                   VALUE 45.
88   CM-CMSCT                    VALUE 46.
88   CM-CMSCU                    VALUE 47.
88   CM-CMSDT                    VALUE 48.
88   CM-CMSED                    VALUE 49.
88   CM-CMSEND                   VALUE 50.
88   CM-CMSERR                   VALUE 51.
88   CM-CMSF                     VALUE 52.
88   CM-CMSID                    VALUE 53.
88   CM-CMSLD                    VALUE 54.
88   CM-CMSLTP                   VALUE 55.
88   CM-CMSMN                    VALUE 56.
88   CM-CMSNDX                   VALUE 57.
88   CM-CMSPDP                   VALUE 58.
88   CM-CMSPLN                   VALUE 60.
88   CM-CMSPM                    VALUE 61.
88   CM-CMSPTR                   VALUE 62.
88   CM-CMSQCF                   VALUE 63.
88   CM-CMSQPM                   VALUE 64.
88   CM-CMSRC                    VALUE 65.
88   CM-CMSRT                    VALUE 66.
88   CM-CMSSL                    VALUE 67.
88   CM-CMSSRM                   VALUE 68.
88   CM-CMSST                    VALUE 69.
88   CM-CMSTC                    VALUE 70.
88   CM-CMSTPN                   VALUE 71.
88   CM-CMTRTS                   VALUE 72.
88   CM-CMWAIT                   VALUE 73.
88   CM-CMWCMP                   VALUE 74.
88   CM-CMSJT                    VALUE 75.
```

```
*
 01  COMPLETED-OP-COUNT           PIC 9(9) COMP-5.
*
 01  CONFIRMATION-URGENCY         PIC 9(9) COMP-5.
     88  CM-CONFIRMATION-NOT-URGENT        VALUE 0.
     88  CM-CONFIRMATION-URGENT            VALUE 1.
*
*  CONTROL-INFORMATION-RECEIVED  USES REQUEST-SEND-RECEIVED  VALUES
*
 01  CONVERSATION-ID              PIC X(8).
*    8 BYTES
*
 01  CONVERSATION-QUEUE           PIC 9(9) COMP-5.
     88  CM-INITIALIZATION-QUEUE           VALUE 0.
     88  CM-SEND-QUEUE                     VALUE 1.
     88  CM-RECEIVE-QUEUE                  VALUE 2.
     88  CM-SEND-RECEIVE-QUEUE             VALUE 3.
     88  CM-EXPEDITED-SEND-QUEUE           VALUE 4.
     88  CM-EXPEDITED-RECEIVE-QUEUE        VALUE 5.
*
 01  CONVERSATION-STATE           PIC 9(9) COMP-5.
     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.
     88  CM-SEND-ONLY-STATE                VALUE 15.
     88  CM-RECEIVE-ONLY-STATE             VALUE 16.
     88  CM-SEND-RECEIVE-STATE             VALUE 17.
     88  CM-PREPARED-STATE                 VALUE 18.
*
 01  CONVERSATION-TYPE            PIC 9(9) COMP-5.
     88  CM-BASIC-CONVERSATION             VALUE 0.
     88  CM-MAPPED-CONVERSATION            VALUE 1.
*
 01  CONVERSATION-SECURITY-TYPE   PIC 9(9) COMP-5.
     88  CM-SECURITY-NONE                  VALUE 0.
     88  CM-SECURITY-SAME                  VALUE 1.
     88  CM-SECURITY-PROGRAM               VALUE 2.
     88  CM-SECURITY-PROGRAM-STRONG        VALUE 5.
*
 01  CM-RETCODE                   PIC 9(9) COMP-5.
*    ===> RETURN-CODE  IS A RESERVED WORD IN SOME  <===
*    ===> VERSIONS  OF  COBOL                      <===
*
*******   THIS PARAMETER ALSO USED FOR ***************
*         CONVERSATION-RETURN-CODE  PARAMETER
*
     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-SYS     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-CONVERSATION-ENDING            VALUE 38.
           88   CM-SEND-RCV-MODE-NOT-SUPPORTED    VALUE 39.
           88   CM-BUFFER-TOO-SMALL               VALUE 40.
           88   CM-EXP-DATA-NOT-SUPPORTED         VALUE 41.
           88   CM-DEALLOC-CONFIRM-REJECT         VALUE 42.
           88   CM-ALLOCATION-ERROR               VALUE 43.
           88   CM-RETRY-LIMIT-EXCEEDED           VALUE 44.
           88   CM-NO-SECONDARY-INFORMATION       VALUE 45.
           88   CM-SECURITY-NOT-SUPPORTED         VALUE 46.
           88   CM-CALL-NOT-SUPPORTED             VALUE 48.
           88   CM-PARM-VALUE-NOT-SUPPORTED       VALUE 49.
           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.
           88   CM-CONV-DEALLOC-AFTER-SYNCPT      VALUE 136.
           88   CM-INCLUDE-PARTNER-REJECT-BO      VALUE 137.
       *
       *
        01  DATA-RECEIVED                PIC 9(9) COMP-5.
           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-5.
           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-5.
     88   CM-RECEIVE-ERROR                VALUE 0.
     88   CM-SEND-ERROR                   VALUE 1.
*
*  EXPEDITED-RECEIVE-TYPE  USES RECEIVE-TYPE VALUES
*
 01  FILL                         PIC 9(9) COMP-5.
     88   CM-FILL-LL                      VALUE 0.
     88   CM-FILL-BUFFER                  VALUE 1.
*
 01  INITIALIZATION-DATA          PIC X(10000).
*    0-10000 BYTES
*
 01  INITIALIZATION-DATA-LENGTH   PIC 9(9) COMP-5.
*
 01  JOIN-TRANSACTION             PIC 9(9) COMP-5.
     88  CM-JOIN-IMPLICIT                 VALUE 0.
     88  CM-JOIN-EXPLICIT                 VALUE 1.
*
 01  LOG-DATA                     PIC X(512).
*    0-512 BYTES
*
 01  LOG-DATA-LENGTH              PIC 9(9) COMP-5.
*
 01  MAXIMUM-BUFFER-SIZE          PIC 9(9) COMP-5.
*
 01  MODE-NAME                    PIC X(8).
*    0-8 BYTES
*
 01  MODE-NAME-LENGTH             PIC 9(9) COMP-5.
*
 01  OOID                         PIC 9(9) COMP-5.
*
 01  PARTNER-LU-NAME              PIC X(17).
*    1-17 BYTES
*
 01  PARTNER-LU-NAME-LENGTH       PIC 9(9) COMP-5.
*
 01  PREPARE-DATA-PERMITTED       PIC 9(9) COMP-5.
     88  CM-PREPARE-DATA-NOT-PERMITTED     VALUE 0.
     88  CM-PREPARE-DATA-PERMITTED         VALUE 1.
*
 01  PREPARE-TO-RECEIVE-TYPE      PIC 9(9) COMP-5.
     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-5.
     88   CM-BLOCKING                     VALUE 0.
     88   CM-NON-BLOCKING                 VALUE 1.
*
*  QUEUE-PROCESSING-MODE  USES PROCESSING-MODE  VALUES
*
 01  RECEIVED-LENGTH              PIC 9(9) COMP-5.
*
```

```
 01  RECEIVE-TYPE                 PIC 9(9) COMP-5.
     88   CM-RECEIVE-AND-WAIT              VALUE 0.
     88   CM-RECEIVE-IMMEDIATE             VALUE 1.
*
 01  REQUESTED-LENGTH             PIC 9(9) COMP-5.
*
 01  REQUEST-TO-SEND-RECEIVED     PIC 9(9) COMP-5.
     88  CM-REQ-TO-SEND-NOT-RECEIVED       VALUE 0.
     88  CM-NO-CONTROL-INFO-RECEIVED       VALUE 0.
     88  CM-REQ-TO-SEND-RECEIVED           VALUE 1.
     88  CM-ALLOCATE-CONFIRMED             VALUE 2.
     88  CM-ALLOCATE-CONFIRMED-DATA        VALUE 3.
     88  CM-ALLOCATE-REJECTED-WITH-DATA    VALUE 4.
     88  CM-EXPEDITED-DATA-AVAILABLE       VALUE 5.
     88  CM-RTS-RCVD-AND-EXP-DATA-AVAIL    VALUE 6.
*
*
 01  RETURN-CONTROL               PIC 9(9) COMP-5.
     88   CM-WHEN-SESSION-ALLOCATED        VALUE 0.
     88   CM-IMMEDIATE                     VALUE 1.
*
 01  SECURITY-PASSWORD            PIC X(10).
*    0-10 BYTES
*
 01  SECURITY-PASSWORD-LENGTH     PIC 9(9) COMP-5.
*
 01  SECURITY-USER-ID             PIC X(10).
*    0-10 BYTES
*
 01  SECURITY-USER-ID-LENGTH      PIC 9(9) COMP-5.
*
 01  SEND-LENGTH                  PIC 9(9) COMP-5.
*
 01  SEND-RECEIVE-MODE            PIC 9(9) COMP-5.
     88   CM-HALF-DUPLEX                   VALUE 0.
     88   CM-FULL-DUPLEX                   VALUE 1.
*
 01  SEND-TYPE                    PIC 9(9) COMP-5.
     88   CM-BUFFER-DATA                   VALUE 0.
     88   CM-SEND-AND-FLUSH                VALUE 1.
     88   CM-SEND-AND-CONFIRM              VALUE 2.
     88   CM-SEND-AND-PREP-TO-RECEIVE      VALUE 3.
     88   CM-SEND-AND-DEALLOCATE           VALUE 4.
*
 01  STATUS-RECEIVED              PIC 9(9) COMP-5.
     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.
     88  CM-TAKE-COMMIT-DATA-OK            VALUE 8.
     88  CM-TAKE-COMMIT-SEND-DATA-OK       VALUE 9.
     88  CM-TAKE-COMMIT-DEALLOC-DATA-OK    VALUE 10.
     88  CM-PREPARE-OK                     VALUE 11.
     88  CM-JOIN-TRANSACTION               VALUE 12.
*
```

```
 01  SYNC-LEVEL                    PIC 9(9) COMP-5.
     88   CM-NONE                            VALUE 0.
     88   CM-CONFIRM                         VALUE 1.
     88   CM-SYNC-POINT                      VALUE 2.
     88   CM-SYNC-POINT-NO-CONFIRM           VALUE 3.
*
 01  SYM-DEST-NAME                 PIC X(8).
*
 01  TIMEOUT                       PIC 9(9) COMP-5.
*
 01  TP-NAME                       PIC X(64).
*    1-64 BYTES
*
 01  TP-NAME-LENGTH                PIC 9(9) COMP-5.
*
 01  TRANSACTION-CONTROL           PIC 9(9) COMP-5.
     88  CM-CHAINED-TRANSACTIONS            VALUE 0.
     88  CM-UNCHAINED-TRANSACTIONS          VALUE 1.
*
 01  USER-FIELD                    PIC X(8).
*
* ***************** END OF PSEUDONYMS *****************
```

Appendix F

Sample Programs

This appendix contains sample programs.

- Section F.1 on page 514

  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.

- Section F.2 on page 518

  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.

- Section F.3 on page 523 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 depend on the nature of actual applications.

F.1 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.
*****************************************************************
```

F.2 CREDRPT (Acceptor of the Conversation)

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.        CREDRPT.
*************************************************************
* THIS IS THE CREDRPT PROGRAM THAT RECEIVES 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-5.
     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.
   ****************************************************************
```

F.3 Results of Successful Program Execution

SALESRPT Program

```
SALESRPT CONVERSATION INITIALIZED
SALESRPT CONVERSATION ALLOCATED
SALESRPT DATA RECORD SENT
SALESRPT RESULTS RECORD RECEIVED
PROGRAM: SALESRPT EXECUTION COMPLETE
```

CREDRPT Program

```
CREDRPT CONVERSATION ACCEPTED
CREDRPT RECORD RECEIVED
 0010   XYZ INC.                    5000.00          724.50
CREDRPT DATA PROCESSED
CREDRPT RESULTS RECORD SENT
CREDRPT DEALLOCATED CONVERSATION
PROGRAM: CREDRPT EXECUTION COMPLETE
```

Appendix G

Application Migration from CPI-C to CPI-C, Version 2

This appendix describes the application migration from the original version of X/Open CPI-C (see the referenced **CPI-C CAE** specification) to Version 2 of CPI-C defined by this specification.

The following points should be noted with regard to migration between these two versions:

- The Accept_Conversation (CMACCP) call behaves differently when no incoming conversation exists. CPI-C Version 2 returns CM_PROGRAM_STATE_CHECK, whereas the original X/Open CPI-C returns CM_OPERATION_INCOMPLETE and a *conversation_ID*. Programs that want to accept multiple conversations should use the following calls instead of Accept_Conversation:

  — Initialize_For_Incoming (CMINIC)

  — Set_Processing_Mode (CMSPM), processing_mode = CM_NON_BLOCKING

  — Accept_Incoming (CMACCI).

  As examples, flows 7 and 8 on pages 34 to 37 of the original CPI-C must be changed.

- All conversation-specific calls except Cancel_Conversation (CMCANC) return a CM_OPERATION_NOT_ACCEPTED return code, instead of CM_PROGRAM_STATE_CHECK, when the conversation-specific call has been attempted after receiving a CM_OPERATION_INCOMPLETE *return_code* value to the previous conversation-specific call and prior to calling Wait_For_Conversation (CMWAIT).

- The Extract_Conversation_Security_User_ID (CMECSU) call is not supported in CPI-C Version 2; the function is available using Extract_Security_User_ID (CMESUI). The name has been changed to force an awareness of the increased length of the *security_user_ID*.

- The Convert_Incoming (CMCNVI) and Convert_Outgoing (CMCNVO) calls may receive an unexpected *return_code* of CM_PROGRAM_PARAMETER_CHECK if the *buffer_length* exceeds the maximum length permitted by the local implementation.

- The CM_SYNC_LEVEL_NOT_SUPPORTED_PGM *return_code* value must be changed to CM_SYNC_LVL_NOT_SUPPORTED_PGM. This was a printing error in the original X/Open CPI-C specification.

- The function specify_Local_TP_Name must be changed to Specify_Local_TP_Name. This was a printing error in the original X/Open CPI-C specification.

- The original X/Open CPI-C specification states that several processes may share the same *conversation_ID* (see page 42 of that specification). This feature is not supported by X/Open CPI-C Version 2.

- The original X/Open CPI-C defines all functions as type CM_RETCODE, for example:

  ```
  extern CM_RETCODE CMACCP;
  ```

  CPI-C Version 2 defines all functions as type **void**, for example:

  ```
  extern CM_ENTRY cmaccp;
  ```

  Programs that test the return value of a CPI-C call must test the parameter *cm_return_code*.

- The readable macros (for example Accept_Conversation instead of CMACCP) are only available in CPI-C Version 2 if the program or the include file contains the line:

```
#define READABLE_MACROS
```

- Some parameters have other types in X/Open CPI-C and CPI-C Version 2. Some compilers may give out warnings when compiling existing CPI-C programs with the new CPI-C Version 2 *include* file.

Table G-1 Comparison of Parameters between X/Open CPI-C Versions

	Original X/Open CPI-C	**X/Open CPIC, Version 2**
Conversation ID parameter	`CONVERSATION_ID (char [8])`	`unsigned char CM_PTR (unsigned char *)`
Character pointers	`char *`	`unsigned char CM_PTR (unsigned char *)`
Length parameters	`int *`	`CM_INT32 CM_PTR (signed long int *)`
Definitions of return codes and numeric parameters	`typedef enum`	`#define`

Glossary

API
Application Programming Interface.

application-entity
The part of an application-process that exclusively defines communication formats and protocols for OSI-compliant systems.

application-entity-qualifier
The qualifier that is used to identify a specific instance of an application-entity. It must be unambiguous within the scope of the application-process. See also ISO / IEC 7498-3.

application-process
The part of an open system that performs the information processing for a particular application. See also ISO / IEC 7498.

application-process-title
The unambiguous title of the application-process. It must be unambiguous within the OSI environment. See also ISO / IEC 7498-3.

application context
The set of rules that define the exchange of information between two application programs. See also ISO / IEC 9545.

application context name
The registered name of the application context. See also ISO / IEC 9545.

association
A relationship between two application-entity instances for the purpose of exchanging data. An association is similar to an SNA LU 6.2 session and is sometimes called a logical connection. See also ISO / IEC 9594.

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 (or thread) is blocked (unable to perform any other work) until the call operation is completed.

callback function
An application-defined function that is called when an outstanding operation completes.

chained transactions
A series of transactions in which the (n+1)th transaction begins immediately upon the termination of the nth transaction. See also ISO / IEC 10026-1.

communication resource manager (CRM)
The component within a system that manages a particular resource — in this case, a conversational communication resource. See also the referenced **DTP** guide.

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 queue
A logical grouping of CPI Communications calls on a conversation. Calls associated with a specific queue are processed serially. Calls associated with different queues are processed independently.

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.

distinguished name
A completely qualified name that is used to access an entry in a distributed directory.

initialization data
Application-specific data that may be exchanged between two application programs during conversation initialization. See also the ISO/IEC 10026-2 standard for User-Data on TP-BEGIN-DIALOGUE and the SNA Transaction Programmers Reference Manual for LU Type 6.2.

local program
The program being discussed within a particular context. Contrast with remote program.

logical connection
The generic term used to refer to either an SNA LU 6.2 session or an OSI association.

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.

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 logical connection that will be allocated for a conversation.

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.

OSI TP
Refers to the International Standard ISO/IEC 10026, Information Technology — Open Systems Interconnection — Distributed Transaction Processing. ISO/IEC 10026 is one of a set of standards produced to facilitate the interconnection of computer systems.

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.

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.

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.

secondary information
Information associated with the return code at the completion of a call. The information can be used to determine the cause of the return code.

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.

subordinate program
The application program that issued either Accept_Conversation or Accept_Incoming for a protected conversation.

superior program
The application program that issued Initialize_Conversation for a protected conversation.

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. Synonymous with transaction manager.

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.

transaction
A related set of operations that are characterized by the ACID (atomicity, consistency, isolation, and durability) properties. See also ISO / IEC 10026-1.

transaction manager (TM)
The component within a system that manages the coordination of resources within a transaction. Synonymous with sync point manager. See also the referenced **DTP** guide.

transition
See state transition.

unchained transactions
A series of transactions in which the (n+1)th transaction does not begin immediately upon the termination of the nth transaction, but is explicitly started at a later time. See also ISO / IEC 10026-1.

user field
Application data that can be associated with an outstanding operation. The data, as specified by the program, can be returned to the program through a Wait_For_Completion call issued for that outstanding operation, or it can be passed to the callback function associated with the outstanding operation, when the operation completes.

X/Open
X / Open is an independent, worldwide, open systems organization whose mission is to bring users greater value from computing, through the practical implementation of open systems.

X/Open TX (Transaction Demarcation) interface
X / Open Distributed Transaction Processing: The TX (Transaction Demarcation) Specification defines an interface between the transaction manager and the application program. It is similar to the IBM SAA resource recovery interface.

Index