IBM Communications Server for AIX

CPI-C Programmer's Guide

V6.3

SC31-8591-02

IBM Communications Server for AIX

CPI-C Programmer's Guide

V6.3

SC31-8591-02

Note:
Before using this information and the product it supports, be sure to read the general information under Appendix D, "Notices," on page 181.

Third Edition (November 2005)

This edition applies to IBM Communications Server for AIX, Version 6.3, program number 5765-E51, and to all subsequent releases and modifications until otherwise indicated in new editions or technical newsletters.

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

IBM welcomes your comments. You may send your comments to the following address:

International Business Machines Corporation
Attn: z/OS Communications Server Information Development
Department AKCA, Building 501
P.O. Box 12195, 3039 Cornwallis Road
Research Triangle Park, North Carolina
27709-2195
U.S.A.

You can send us comments electronically by using one of the following methods:

- Fax (USA and Canada): 1-919-254-4028
- Internet e-mail: comsvrcf@us.ibm.com

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Tables

Figures

About This Book

This book is a guide for developing C-language or Java®application programs that use Common Programming Interface for Communications (CPI-C) to exchange data in a Systems Network Architecture (SNA) environment. Communications Server for AIX (hereafter referred to as CS/AIX) is an IBM®software product that enables a server running AIX®to exchange information with other nodes on an SNA network.

The CS/AIX implementation of CPI-C is based on IBM's implementation of CPI-C in its OS/2®products (with modifications for the AIX environment).

Programs written to use the CS/AIX implementation of CPI-C can exchange data with programs written to use other implementations of CPI-C that adhere to the SNA Logical Unit (LU) 6.2 architecture.

This book applies to V6.3 of CS/AIX running on AIX Version 5.2 and higher base operating system.

To submit comments and suggestions about *Communications Server for AIX CPI-C Programmer's Guide*, use the Reader's Comment Form located at the back of this book. This form provides instructions for submitting your comments by mail, by FAX, or by electronic mail.

Who Should Use This Book

This book is intended for experienced C or Java programmers who write Systems Network Architecture (SNA) transaction programs for systems with CS/AIX. Programmers may or may not have prior experience with SNA or the communication facilities of CS/AIX.

Application programmers design and code transaction and application programs that use the CS/AIX programming interfaces to send and receive data over an SNA network. They should be thoroughly familiar with SNA, the remote program with which the transaction or application program communicates, and the AIX or Linux operating system programming and operating environments.

More detailed information about writing application programs is provided in the manual for each API or (for back-level APIs) in *Communications Server for AIX Transaction Program Reference V4R2* (SC31–8212). For additional information about CS/AIX publications, see the bibliography.

How to Use This Book

This section explains how information is organized and presented in this book.

Organization of This Book

This book is organized as follows:

- Chapter 1, "Concepts," on page 1, introduces the fundamental concepts of CPI-C. It is intended for programmers who are not familiar with CPI-C.
- Chapter 2, "Writing CPI-C Applications," on page 19, contains general information a CPI-C programmer needs when writing CPI-C Applications.

- Chapter 3, "CPI-C Calls," on page 47, describes each CPI-C call in detail. Each description includes the following: purpose, parameters, conversation states in which the call can be issued, and conversation state changes after the call has executed. Differences between the implementations of CPI-C for the different operating systems are indicated where they occur.
- Chapter 4, "Sample CPI-C Transaction Programs," on page 163, describes the CS/AIX CPI-C sample programs that illustrate the use of CPI-C calls in a C program, and includes instructions for compiling, linking, and running the programs.

  AIX, LINUX

- Chapter 5, "Sample Java CPI-C Transaction Program," on page 167, describes the CS/AIX Java CPI-C sample program that illustrates the use of CPI-C calls in a Java application, and includes instructions for compiling, linking, and running the program.

  

- Appendix A, "Return Code Values," on page 169, lists all the possible return codes in the CPI-C interface in numerical order and gives their meanings.
- Appendix B, "Common Return Codes," on page 171, documents certain return codes that are common to several calls.
- Appendix C, "Conversation State Changes," on page 177, provides information about CPI-C conversation states: which CPI-C calls are permitted in each state, and the state to which the conversation changes on return from each call.

Typographic Conventions

Table 1 shows the typographic styles used in this document.

Table 1. Typographic Conventions

Special Element	Sample of Typography
Document title	*Communications Server for AIX Administration Guide*
File or path name	**cmc.h**
Command or AIX / Linux utility	**vi**
Option or flag	**-I**
Parameter or Motif field	*data_received*; *request_to_send_received*
Literal value or selection that the user can enter (including default values)	`0`; `32,767`
Constant or signal	`CM_NONE`
Return value	`CM_OK`; `CM_PRODUCT_SPECIFIC_ERROR`
Variable representing a supplied value	*functionname*
Environment variable	`APPCTPN`
Programming verb	RECEIVE
User input	**cc -I**
Function, call, or entry point	`WinCPICSetEvent`
Data structure	`WCPICDATA`
Hexadecimal value	`0x20`

Graphic Conventions

AIX, LINUX

This symbol is used to indicate the start of a section of text that applies only to the AIX or Linux operating system. It applies to AIX servers and to the IBM Remote API Client running on AIX, Linux, Linux for pSeries or Linux for zSeries.

This symbol is used to indicate the start of a section of text that applies to the IBM Remote API Client on Windows.

This symbol indicates the end of a section of operating system specific text. The information following this symbol applies regardless of the operating system.

What's New

Communications Server for AIX V6.3 replaces Communications Server for AIX V6.1.

Releases of this product that are still supported are:

- Communications Server for AIX V6.1

The following releases of this product are no longer supported:

- Communications Server for AIX Version 6 (V6)
- Communications Server for AIX Version 5 (V5)
- Communications Server for AIX Version 4 Release 2 (V4R2)
- Communications Server for AIX Version 4 Release 1 (V4R1)
- SNA Server for AIX Version 3 Release 1.1 (V3R1.1)
- SNA Server for AIX Version 3 Release 1 (V3R1)
- AIX SNA Server/6000 Version 2 Release 2 (V2R2)
- AIX SNA Server/6000 Version 2 Release 1 (V2R1) on AIX 3.2
- AIX SNA Services/6000 Version 1

Where to Find More Information

See the bibliography for other books in the CS/AIX library, as well as books that contain additional information about topics related to SNA and AIX workstations.

The information in the CS/AIX books is also available in HTML format. You can use this library to search for specific information or to view online versions of each of the CS/AIX books.

Chapter 1. Concepts

This chapter introduces the fundamental concepts of CPI-C in a distributed processing environment. The following topics are covered:

- What is CPI-C?
- An example of a simple mapped conversation
- Confirmation processing
- Conversation states
- How to change conversation states
- Side information
- Basic conversations
- Multiple conversations
- Conversation security
- Nonblocking operation
- CPI-C and LU 6.2

What Is CPI-C?

CPI-C stands for Common Programming Interface for Communications. CPI-C is a portable application programming interface, or API, that enables peer-to-peer communications among programs in an SNA environment.

CPI-C enables application programs distributed across a network to work together. By communicating with each other and exchanging data, they can accomplish a single processing task, such as querying a remote data base, copying a remote file, or sending or receiving electronic mail.

These programs communicate as peers, on an equal (rather than hierarchical) basis. Together, programs distributed across a local-area or wide-area network perform distributed processing.

CS/AIX CPI-C Option Set Support

AIX, LINUX

For C programs (not for Java programs), CS/AIX CPI-C implements IBM's CPI-C 2.0. It supports the mandatory CPI-C 2.0 conformance class, Conversations, and the following optional conformance classes:

- LU 6.2
- Conversation-level nonblocking operation
- Server
- Data conversion routines
- Security

In addition, CS/AIX CPI-C provides support for additional functions that were defined as part of the X/Open CPI-C implementation and have been incorporated into IBM CPI-C 2.0. CS/AIX supports these entry points for back-compatibility with existing CPI-C applications. Wherever possible, CPI-C programmers should

use the IBM CPI-C 2.0 versions of the functions. The mapping between the X/Open functions and the IBM CPI-C 2.0 functions is shown in Table 2.

Table 2. Mapping Between X/Open Functions and IBM CPI-C 2.0 Functions

X/Open Function	IBM CPI-C 2.0 Function
Extract_Conversation_Security_User_ID (xcecsu)	Extract_Security_User_ID (cmesui)
Set_Conversation_Security_Password (xcscsp)	Set_Conversation_Security_Password (cmscsp)
Set_Conversation_Security_Type (xcscst)	Set_Conversation_Security_Type (cmscst)
Set_Conversation_Security_User_ID (xcscsu)	Set_Conversation_Security_User_ID (cmscsu)

For Java programs, CS/AIX implements Java CPI-C as in IBM's CS/Windows product (the package COM.ibm.eNetwork.cpic). It also includes three additional CPI-C functions (Set_Conversation_Context, Set_Local_LU_Name, and Extract_Local_LU_Name) which are part of the standard CS/AIX CPI-C implementation but are not included in CS/Windows.

WINDOWS

CS/AIX CPI-C on Windows implements Windows CPI-C (as defined by the WOSA SNA specification).

Communication between Programs

Many hardware and software elements in the SNA environment are required in order for two programs to communicate with each other. The following diagram illustrates the elements relevant to programmers.

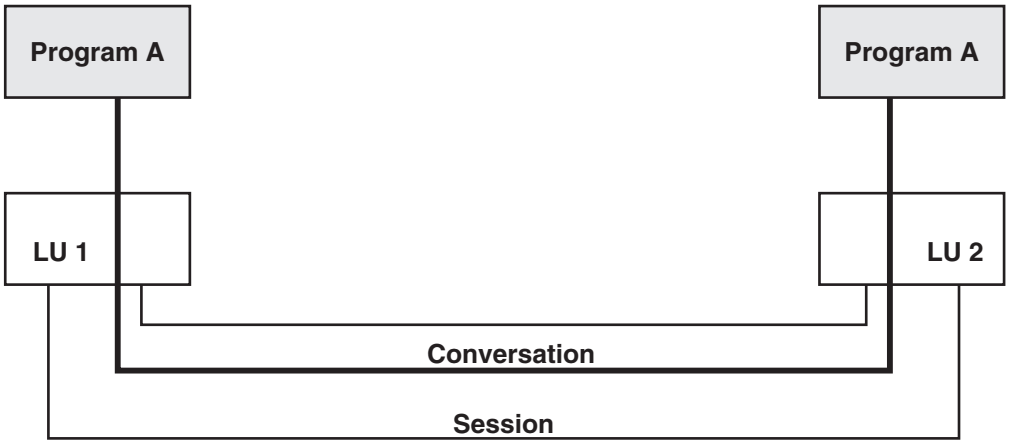

Figure 1. Communication between Programs

Logical Unit 6.2

Each program is associated with a logical unit (LU), which is the program's access point into the network. CPI-C uses LU type 6.2, which supports peer-to-peer communications between LUs. Several programs can be associated with the same LU.

Sessions

Before two programs can communicate, their LUs must be connected through an LU-to-LU session—a logical connection between the two LUs. The session is established using a particular mode—a set of networking characteristics that determines how the LUs use the session.

An LU type 6.2 can have multiple sessions (two or more concurrent sessions with different partner LUs) and parallel sessions (two or more concurrent sessions with the same partner LU). During configuration, the System Administrator or user determines how many sessions a particular LU supports and whether the LU supports parallel sessions.

Conversations

The communication between the two programs occurs as a conversation within the LU-to-LU session. A program can be involved in several conversations simultaneously.

Contention

When both LUs attempt to allocate a conversation on the same session at the same time, one must win (the contention winner) and one must lose (the contention loser). The mode used by the two LUs specifies the number of contention winner and contention loser sessions for each LU; the contention winner LU and the contention loser LU are determined when the session is established.

In a session, the contention loser LU must ask permission from the contention winner LU before allocating a conversation. The contention winner may or may not grant permission. The contention winner LU, on the other hand, simply allocates a conversation when desired.

Characteristics

A conversation has a set of internal values that control the overall operation of the conversation and the behavior of individual calls. These values are called characteristics.

CPI-C Calls

A program accesses CPI-C through CPI-C calls. Each call performs a particular action such as starting or ending a conversation, sending or receiving data, setting an option that determines how subsequent CPI-C calls will operate, or obtaining information about the options currently in use. On each call, the program supplies parameters to CPI-C, which performs the requested function and returns new parameters to the program.

The program issuing the call is referred to as the local program; the other program is referred to as the partner program. Similarly, the LU serving the local program is the local LU; the LU serving the partner program is the partner LU.

Programs and LUs residing on other nodes in the network are also called remote programs and remote LUs.

The Conversation Process

A conversation begins when both of the following happen:

- One program (the invoking program) instructs CS/AIX to start another program (the invoked program) and allocate a conversation between the two programs.

- The invoked program notifies CS/AIX that it is ready to communicate with the invoking program.

During the conversation, the two programs exchange status information and application data. Typically, a conversation ends when a program instructs CS/AIX to deallocate the conversation.

Conversation Types

A conversation can be mapped or basic.

In general, mapped conversations are used by application programs. These are programs that accomplish tasks for end users. Mapped conversations are less complex than basic conversations. In a mapped conversation, the programs send and receive data one record at a time.

Basic conversations are normally used by service programs. These are programs that provide services to other local programs. Basic conversations provide an experienced LU 6.2 programmer with a greater degree of control over the transmission and handling of data. For further information, see "Basic Conversations" on page 11.

A Simple Mapped Conversation

The example below charts a simple mapped conversation. It shows the CPI-C calls used to start the conversation, exchange data, and end it. The arrow indicates the flow of data. Some call parameters and some return codes are also shown, enclosed in parentheses.

Table 3. A Simple Mapped Conversation

Invoking Program		Invoked Program
Initialize_Conversation Allocate Send_Data Deallocate	→	Accept_Conversation Receive (*data_received*=CM_COMPLETE_DATA_RECEIVED) (*return_code*=CM_DEALLOCATED_NORMAL)

Starting a Conversation

To start a conversation, the invoking program issues the following calls:

- Initialize_Conversation, which requests CPI-C to set the characteristics of the conversation.

  The Initialize_Conversation call specifies a symbolic destination name, which is associated with a CPI-C side information entry in the CS/AIX configuration. This entry specifies partner program, partner LU, mode, and security information.
- Allocate, which requests that CS/AIX establish a conversation between the invoking program and the invoked program.

The invoked program issues the Accept_Conversation call, which informs CS/AIX that the invoked program is ready to begin a conversation with the invoking program.

Sending Data

The Send_Data call puts one data record (containing application data to be transmitted) into the local LU's send buffer which already contains the allocation request. The transmission of the data to the partner program does not happen until one of the following events occurs:

- The send buffer fills up
- The program issues a call that forces CS/AIX to flush the buffer (and send the data to the partner program)

The Deallocate call flushes the send buffer sending the allocation request and data to the partner program.

Receiving Data

The Receive call receives the data record and status information from the partner program. If no data or status information is currently available, the local program, by default, waits for data to arrive.

The *data_received* parameter of the Receive call tells the program whether it received data and if so, whether the data is complete or not.

Ending a Conversation

To end a conversation, one of the programs issues the Deallocate call, which causes CS/AIX to deallocate the conversation between the two programs.

Confirmation Processing

When a program sends data to its partner program, it can also request the partner program to confirm that it has received the data successfully. The receiving program must either confirm receipt of the data or indicate that an error has occurred. The two programs are synchronized each time they exchange a confirmation request and response. This is illustrated in Table 4.

Table 4. Confirmation Processing

Invoking Program		Invoked Program
Initialize_Conversation Set_Sync_Level (*sync_level*=CM_CONFIRM) Allocate Send_Data Confirm	→	
		Accept_Conversation Receive (*data_received*=CM_COMPLETE_DATA_RECEIVED) (*status_received*=CM_CONFIRM_RECEIVED) Confirmed
	←	
(*return_code*=CM_OK) Send_Data		

Table 4. Confirmation Processing (continued)

Invoking Program		Invoked Program
Deallocate		
	→	
		Receive (*status_received*=CM_CONFIRM_DEALLOC_RECEIVED) Confirmed
	←	
(*return_code*=CM_OK)		

Establishing the Synchronization Level

The synchronization level is one of the conversation's characteristics. There are two possible synchronization levels:

- CM_NONE, the default, under which confirmation processing does not occur
- CM_CONFIRM, under which the programs can request confirmation of receipt of data and respond to such requests

The default synchronization level is CM_NONE; you can override this using the Set_Sync_Level call.

Sending a Confirmation Request

Issuing the Confirm call does the following:

- It flushes the local LU's send buffer (which sends any data contained in the buffer to the partner program)
- It sends a confirmation request, which the partner program receives through the *status_received* parameter of a Receive call

After issuing the Confirm call, the invoking program waits for confirmation from the invoked program.

Receiving a Confirmation Request

The *status_received* parameter of the Receive call indicates any future action required by the local program.

In the previous example, the first Received call has a *status_received* of CM_CONFIRM_RECEIVED, indicating that a confirmation is required before the partner program can continue.

Responding to a Confirmation Request

The invoked program issues the Confirmed call to confirm receipt of data; this frees the invoking program to resume processing.

Deallocating the Conversation

Because the synchronization level of the conversation is set to CM_CONFIRM, the Deallocate call sends a confirmation request with the data flushed from the buffer.

For the second Receive call, *status_received* is CM_CONFIRM_DEALLOC_RECEIVED, indicating that the partner program requires a confirmation, generated by the Confirmed call, before the conversation can be deallocated.

Conversation States

The state of the conversation governs which CPI-C calls can be issued by the program. For instance, a program cannot issue the Send_Data call if the conversation is not in Send or Send-Pending state. Possible conversation states are summarized in the list below.

Reset The conversation has not started or has been terminated.

Initialize
The conversation has been initialized successfully.

Send The program can send data to the partner program and request confirmation. When the conversation is in Send state, the program can also begin to receive data, which can cause the state to change to Receive.

Send-Pending
The program issued a Receive call and received data as well as a send indicator (*status_received* = `CM_SEND_RECEIVED`), indicating that the program can begin to send data. This state is similar to Send state, except that the program can provide additional information when reporting errors (to indicate whether it detected an error in the received data or in its own processing).

Receive
The program can receive application data and status information from the partner program. When the conversation is in Receive state, the program can also send error information and request permission to send data.

Confirm
The program has received a request for confirmation of receipt of data; it must respond positively or send error information to the partner program.

Confirm-Deallocate
The program has received a request for confirmation and must respond positively or send error information. If the program responds positively, the partner program deallocates the conversation.

Confirm-Send
The program has received a request for confirmation; it must respond positively or send error information. After responding, the program can begin to send data.

AIX, LINUX

Initialize-Incoming
The program has successfully issued Initialize_For_Incoming and obtained a conversation ID. It can now issue Accept_Incoming to accept an incoming conversation.

WINDOWS

Pending-Post
The program has successfully issued the Receive call in nonblocking mode. While the call is outstanding, it can issue a limited range of CPI-C calls on this conversation, issue CPI-C calls on other conversations, or continue with other processing.

The description of each CPI-C call includes information about the conversation states in which it can be issued. For a table of which verbs can be issued in each conversation state, see Appendix C, “Conversation State Changes,” on page 177.

The Program’s View of the Conversation

It is the conversation rather than the program that is in a particular state. A program can be conducting several conversations, each of which is in a different state. If a conversation is said to be in Send state, this is from the viewpoint of the local program. To the partner program, the conversation is in another state (such as Receive).

State Changes

A change in the conversation state can result from any of the following:

- A call issued by the local program
- A call issued by the partner program
- An error condition

State Checks

A state check occurs when a program issues a CPI-C call, and the conversation is not in the appropriate state. For instance, a state check would occur if a program issued the Send_Data call while the conversation was in Receive state. When a state check occurs, CPI-C does not execute the call; it returns state-check information through the *return_code* parameter.

Changing Conversation States

In Table 5, the conversation states appear in the left and right margins. This table shows how CPI-C calls can change the state of the conversation from Send to Receive and from Receive to Send.

Table 5. Changing Conversation States

State	Invoking Program	Invoked Program	State
Reset			
	Initialize_Conversation		
Initialize			
	Set_Sync_Level		
	(*sync_level*=CM_CONFIRM)		
	Allocate		
Send			
	Send_Data		
	Prepare_To_Receive		Reset
		→	
		Accept_Conversation	
			Receive
		Receive	
		(*status_received*=CM_CONFIRM_SEND_RECEIVED)	
			Confirm-Send
		Confirmed	
		←	
			Send
	(*return_code*=CM_OK)		

Table 5. Changing Conversation States (continued)

State	Invoking Program	Invoked Program	State
Receive			
		Send_Data	
		Confirm	
		←	
	Receive		
	(*status_received*=		
	CM_CONFIRM_RECEIVED)		
Confirm			
	Request_To_Send		
	Confirmed		
		→	
Receive			
		(*return_code*=CM_OK)	
		(*request_to_send_received*=	
		CM_REQ_TO_SEND_RECEIVED)	
		Prepare_To_Receive	
		←	
	Receive		
	(*status_received*=		
	CM_CONFIRM_SEND_RECEIVED)		
Confirm-Send			
	Confirmed		
		→	
Send			
		(*return_code*=CM_OK)	
			Receive
	Send_Data		
	Deallocate		
		→	
		Receive	
		(*status_received*=	
		CM_CONFIRM_DEALLOC_RECEIVED)	
			Confirm-Deallocate
		Confirmed	
		←	
			Reset
	(*return_code*=CM_OK)		
Reset			

Initial States

Before the conversation is allocated, both programs are in Reset state.

After the conversation is allocated, the initial state is Send for the invoking program and Receive for the invoked program.

Changing to Receive State

The Prepare_To_Receive call enables a program to change the conversation from Send to Receive state. This call does the following:

- It flushes the local LU's send buffer.
- If the synchronization level is set to CM_CONFIRM, the Prepare_To_Receive call sends a CM_CONFIRM_SEND indicator to the partner program through the

status_received parameter of a Receive call. This indicator tells the partner program that a Confirmed response is expected before the partner program can begin to send data.

Changing to Send State

The conversation state for a program changes from Receive to Send when its partner program begins to receive data (by issuing the Prepare_To_Receive call).

The local program (for which the conversation is in Receive state) can inform the partner program that it wants to send data, by issuing the Request_To_Send call. This request is communicated to the partner program through the *request_to_send_received* parameter. (In the previous example, this parameter is shown on the Confirm call; it is also returned to Send_Data and other calls.)

Issuing the Request_To_Send call does not change the state of the conversation, because the partner program can ignore it. When the partner program issues the Prepare_To_Receive call, the conversation state changes to Receive for the partner program. The local program receives the SEND indication on a subsequent RECEIVE verb, and can then send data.

Side Information

The information required for two programs to communicate is stored in CPI-C side information entries in the CS/AIX configuration file. Each side information entry is identified by a Symbolic Destination Name, which is the *sym_dest_name* parameter specified by the Initialize_Conversation call. The parameter *sym_dest_name* is an 8-byte ASCII character string and can contain any displayable characters. It contains the following fields:

- Partner LU name
- Partner program type and name
- Mode name
- Conversation security type (see “Multiple Conversations” on page 12)
- Security user ID and password required to access the partner program

CPI-C also provides two mechanisms for an application to override the configured side information entries, as follows. Both of these mechanisms apply only to the application’s own use of this information, and do not modify the original version stored in the configuration file.

- The application can use the Set_CPIC_Side_Information, Extract_CPIC_Side_Information, and Delete_CPIC_Side_Information calls to manage its own local copy of complete side information entries. (These functions are not available in Java CPI-C.)
- The application can use CPI-C Set_* functions (such as Set_Partner_LU_Name) to override an individual parameter from the side information before allocating the conversation.

For more information, see “Side Information” on page 30.

Basic Conversations

Basic conversations are normally used by service programs. These are programs that provide services to other local programs. They are more complex than mapped conversations but provide an experienced LU 6.2 programmer with a greater degree of control over the transmission and handling of data. This section summarizes the characteristics of basic conversations.

Logical Records

In a basic conversation, data is sent in the form of logical records. A logical record is a record that has the general data stream (GDS) syntax described in this section. For more information about GDS syntax, refer to *IBM Systems Network Architecture: Formats*.

The sending TP must format the data into multiple logical records, and the receiving TP must decode the logical records into usable data.

If a logical record is a single record, it consists of the following fields:

- A 2-byte record-length (LL) field
- A 2-byte GDS identifier (ID) field (for example, `0x12FF` identifies the data as application data)
- A data field that can range in length from 0 to 32,763 bytes

The first four bytes are called the LLID.

If a logical record has multiple parts, the first part has the same format as a single record, and all subsequent parts consist of the following fields:

- A 2-byte record-length (LL) field
- A data field that can range in length from 0 to 32,765 bytes

The length recorded in the LL field includes the two bytes of the LL field (and the two bytes of the ID field, if it is present). For example, a single part GDS with no data has a value of `0x0004` for its LL field. The LL field must be in high-low format, rather than byte-swapped format. For example, a length of 230 bytes is represented as `0x00E6`, rather than `0xE600`.

Bit 0 of byte 0 of the LL (the most significant bit) is used to indicate length continuation (segmentation). The following example shows ten bytes of data (each data byte has the value `DD`) split into three GDS segments. The first and second segments each contain four bytes of data, and the last segment contains two bytes of data.

```
8008 12FF DDDD DDDD
8006 DDDD DDDD
0004 DDDD
```

The following values for the LL field are not valid:

- `0x0000`
- `0x0001`
- `0x8000`
- `0x8001`

Error Log Data

In case of an error or abend in a basic conversation, a program can send an error message, in the form of a general data stream (GDS) error log variable, to the partner LU.

Multiple Conversations

A program can be involved in several conversations simultaneously. Each conversation requires an LU-to-LU session. Multiple conversations are not supported if the application uses a dependent LU (for more information, see "Specifying the Local LU" on page 34).

A common use of multiple conversations is to have an invoked program invoke another program, which, in turn, invokes another program, and so on. In the diagram below, program A invokes program B; program B invokes program C.

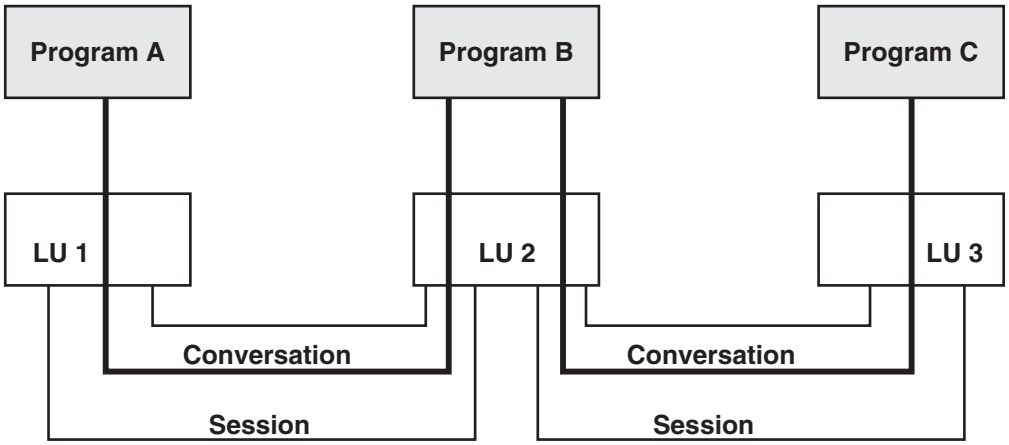

Figure 2. Multiple Conversations

For more information about how CPI-C conversation security operates with multiple conversations, see "Overview of Conversation Security."

Overview of Conversation Security

You can use conversation security to require that the invoking program provide a user ID and password before CPI-C allocates a conversation with the invoked TP.

In configuring the invoked TP, the System Administrator indicates whether to use conversation security. If so, the invoking TP must provide a user ID and password when allocating a conversation with the invoked program. These are either taken from the side information or specified explicitly by the invoking program, and must match a user ID and password configured for the invoked program.

CS/AIX also supports LU-LU session security, which provides security checking when starting the session between the local and remote LUs. LU-LU session security is specified during configuration, and does not require any action in CPI-C programs. For more information, refer to the *Communications Server for AIX Administration Guide*.

Conversation Security for Multiple Conversations

In the example shown in "Multiple Conversations" on page 12, when program A invokes program B and B then invokes C as a result of the conversation with A, the configuration of C may indicate that it will accept an "already-verified" security indication. In this case, the user ID and password supplied by A must still be verified against the configuration for B. However, when B invokes C, it sets the *security_type* conversation characteristic to "same", and CPI-C sends to C the user ID supplied by A and an indication that security has already been verified. For more information, see "Set_Conversation_Security_Type (cmscst)" on page 116.

AIX, LINUX

If the program is involved in more than one pair of incoming and outgoing conversations in this way, it needs to indicate which incoming conversation is to provide the user ID for an outgoing conversation. To do this, CPI-C associates each conversation with a specific "context ID". This is assigned and used as follows:

- Each time the program successfully issues Accept_Conversation or Accept_Incoming, CPI-C assigns a new context ID to the conversation. The program can determine the value of this context ID by issuing Extract_Conversation_Context with the appropriate conversation ID.
- The program's "current context" is normally the context ID associated with the most recent Accept_Conversation or Accept_Incoming. The program can use Set_Conversation_Context to set the current context to the context ID of another of its incoming conversations (subject to the restriction described below).
- Any Allocate call is issued in the program's current context. This means that, if the conversation security type is "same", the user ID from the incoming conversation associated with the current context ID will be sent to the partner program.

In the previous example, program B must ensure that its current context is the context associated with the incoming conversation from program A, before issuing the Allocate call to program C. This ensures that A's user ID is sent on the allocation request to program C. The current context will normally be the correct one, unless B has issued another Accept_Conversation, Accept_Incoming, or Set_Conversation_Context call since accepting the conversation from A.

When a program uses Set_Conversation_Context to change its current context, CS/AIX does not retain the information from the previous context unless there is still at least one active conversation associated with it. This means that, if B finishes the conversation with A and then changes its current context to communicate with a different program, it will not be able to return to the first context ID in order to allocate the conversation with C. If it needs to end the conversation with A before allocating the conversation to C, it must allocate the conversation to C before changing its current context to any other value.

Already-Verified Conversation Security

AIX, LINUX

In some cases, a program may need to indicate "already verified" security when it has not itself been invoked by another program, but has obtained and verified the appropriate security information by another means (for example, by a user entering a user ID and password during a logon sequence). CS/AIX supports this as follows:

- If the program specifying "already verified" was itself invoked by another program, as described in "Conversation Security for Multiple Conversations" on page 13, CPI-C sends the user ID from the current conversation context.
- Otherwise, CPI-C takes the AIX or Linux user name with which the program is running, truncated to 10 characters if necessary, and uses this as the conversation security user ID. Ensure that this name consists of valid AE-string characters and is a valid user name for the program being invoked.
- If the application uses a different method of obtaining the security information (for example, if it requires the user to specify a user ID and password explicitly, rather than relying on the AIX or Linux system security), then it can use either of the CPI-C functions Set_Conversation_Security_User_ID or Set_CPIC_Side_Information to specify this *user_id* to CPI-C before allocating the conversation.

Nonblocking Operation

AIX, LINUX

This section does not apply to Java CPI-C. Java CPI-C functions always operate in blocking mode; that is, the function does not return control to the application until the requested processing has completed.

By default, CPI-C functions operate in blocking mode; that is, the function does not return control to the application until the requested processing has completed. For example, the Confirm function does not return until CPI-C has sent a confirmation request to the partner application and received either an OK or an error response from it.

CPI-C functions can also operate in nonblocking mode; that is, the function returns control to the application immediately, even if the requested processing has not yet completed. This enables the application to continue with other processing that is not related to this conversation, and obtain the results of the verb processing at a later stage.

AIX, LINUX

The application can use the function Check_For_Completion to determine whether a previous nonblocking function has now completed, or Wait_For_Conversation to wait for it to complete. Table 6 on page 15 shows an example of the use of nonblocking mode.

Table 6. Nonblocking Operation

Invoking Program		Invoked Program
Initialize_Conversation		
Allocate		
Send_Data		
Set_Processing Mode (CM_NON_BLOCKING)		
Confirm		
	→	
(*return_code*=CM_OPERATION_INCOMPLETE)		
[Application can perform other processing not related to this conversation.]		Accept_conversation
		Receive
		(*data_received*=CM_COMPLETE_DATA_RECEIVED)
		(*status_received*=CM_CONFIRM_RECEIVED)
Wait_For_Conversation		
[Application is suspended until processing for the previous Confirm has completed]		
		Confirmed
	←	
(Wait_For_Conversation returns, *return_code*=CM_OK, conversation_return_code=CM_OK)		
Send_Data		
Deallocate		
	→	
(*return_code*=CM_OPERATION_INCOMPLETE)		
[Application performs other processing not related to this conversation.]		
		Receive
		(*status_received*=
		CM_CONFIRM_DEALLOC_RECEIVED)
		Confirmed
	←	
Check for Completion		
(*return_code*=CM_OK)		
Wait_For_Conversation		
(*return_code*=CM_OK, conversation_return_code=CM_OK)		
[Conversation is now deallocated.]		

The following steps explain the processing shown in the previous example.

1. After allocating the conversation and sending some data, the invoking program issues Set_Processing_Mode to set the processing mode to CM_NON_BLOCKING. This indicates that subsequent functions on this conversation can operate in nonblocking mode.
2. The invoking program then issues Confirm, which returns CM_OPERATION_INCOMPLETE. This indicates that the function was issued successfully and is operating in nonblocking mode.
3. The program can now perform other processing not related to this conversation, including issuing CPI-C functions on other conversations. It can also issue a limited range of CPI-C functions on this conversation (such as the Extract_* functions). This is different from the IBM CPI-C 2.0 specification, in which the program cannot issue any functions on this conversation other than Wait_For_Conversation or Cancel_Conversation.
4. At some later time, the program issues Wait_For_Conversation to wait for the previous nonblocking function to complete. Since the partner program has not

yet issued Confirmed, processing for the previous Confirm function has not completed, so the invoking program is suspended.

5. When the partner program issues Confirmed, this completes the processing of the invoking program's Confirm function. The Wait_For_Conversation function then returns. The *return_code* of `CM_OK` indicates that Wait_For_Conversation completed successfully; the conversation *return_code* of `CM_OK` indicates that the Confirm function (for which it was waiting) completed successfully.
6. After sending additional data, the invoking program then issues Deallocate, which returns `CM_OPERATION_INCOMPLETE`. This indicates that the function was issued successfully and is operating in nonblocking mode. As before, the program can now perform other processing not related to this conversation, but cannot issue most CPI-C functions on this conversation.
7. The partner program receives the Deallocate request and replies with Confirmed. This completes the processing for the Deallocate function.
8. The invoking program issues Check_For_Completion, to determine whether any previous nonblocking functions on any of its conversations have completed. Since the Deallocate processing has already completed, Check_For_Completion returns with the *conversation_ID* of this conversation.
9. The program then issues Wait_For_Conversation, to get the result of the Deallocate processing. This returns immediately because the Deallocate processing has already completed.

WINDOWS

The application should use the Specify_Windows_Handle function before issuing any verbs in nonblocking mode. This function specifies a Windows handle to which CPI-C sends a message when the verb processing has completed. This message notifies the application that the verb has completed; there is no need for the application to issue an additional call to wait for the results of the verb processing.

CPI-C can use an alternate method to indicate the verb has completed—signaling an event handle. If the application registers an event with the conversation using `WinCPICSetEvent`, then the application can call the Win32 functions `WaitForSingleObject` or `WaitForMultipleObjects` to wait to be notified of the completion of the verb.

If the outstanding call is a Receive call, the application can issue the following calls while Receive is outstanding:

- Request_To_Send
- Send_Error
- Test_Request_to_Send_Received
- Cancel_Conversation
- Deallocate

As an alternative to using Specify_Windows_Handle or `WinCPICSetEvent` as described previously, the application can use Wait_For_Conversation, as for AIX systems. This function is provided for Windows systems to assist in migrating applications from other operating system environments. However, the use of blocking functions such as Wait_For_Conversation in the Windows environment is

strongly discouraged. If you are writing a new application specifically for the Windows environment, use Specify_Windows_Handle and not Wait_For_Conversation.

Note:

- Check_For_Completion, described previously for AIX or Linux systems, is not supported on Windows systems.
- If the application uses one of the calls listed previously in nonblocking mode while Receive is outstanding, it must use Specify_Windows_Handle. It must not issue Wait_For_Conversation if another call is outstanding in addition to Receive; the results of this call are undefined if more than one call is outstanding on the same conversation.

CPI-C and LU 6.2

CPI-C applications can communicate with non-CPI-C LU 6.2 applications, such as APPC.

CPI-C does not support the following features that are included in some LU 6.2 implementations:

- Sync Point/Back Out processing
- PIP data
- LOCKS=LONG
- MAP_NAME
- FMH_DATA

These must not be used in LU 6.2 applications if CPI-C is to communicate with them.

Chapter 2. Writing CPI-C Applications

This chapter contains information you will need when writing CPI-C application programs. The following topics are covered:

- CPI-C call summary
- Initial conversation characteristics
- Side information
- Configuration
- Specifying the TP name and local LU name for a CPI-C program
- How programs get started

AIX, LINUX

- AIX or Linux considerations
- Java CPI-C considerations

WINDOWS

- Windows considerations

- Writing portable applications

CPI-C Call Summary

This section briefly describes each CPI-C call. They are grouped by function. For a more detailed explanation of a particular call, see Chapter 3, "CPI-C Calls," on page 47.

The "names" of the calls are pseudonyms. The actual C function names appear in parentheses after the pseudonym. For example Initialize_Conversation is the pseudonym for a call. The actual function name is cminit.

It may also be necessary to set the local TP and LU names that the program will use. For more information about this, see "Specifying the Local TP Name" on page 33 and "Specifying the Local LU" on page 34.

Starting a Conversation

The following calls are used to start a conversation between two programs. For more information about this subject, see "How Programs Get Started" on page 36.

You may also need to set the local TP name and LU name that the program will use. For information about setting these, see "Specifying the Local TP Name" on page 33 and "Specifying the Local LU" on page 34.

WinCPICStartup

WINDOWS

This call registers the application as a Windows CPI-C application, and determines whether the CPI-C software supports the level of function required by the application. A Windows CPI-C application must use this call before issuing any other CPI-C calls.

Initialize_Conversation (cminit)

This call is issued by the invoking program to obtain a conversation ID and to set the initial values for the conversation's characteristics. The initial values are derived from side information associated with the symbolic destination name, or are CPI-C defaults.

Initialize_For_Incoming (cminic)

This call is used by the invoked program to obtain a conversation ID for an incoming conversation which it will later accept with Accept_Incoming. This enables the program to issue Accept_Incoming in nonblocking mode, if required, instead of using Accept_Conversation which always operates in blocking mode.

Set_* Calls to Change Initial Conversation Characteristics

After issuing the Initialize_Conversation call, the invoking program can change the initial conversation characteristics by issuing any of the calls listed in Table 7. These calls can only be issued in Initialize state.

Table 7. Set_* Calls to Change Initial Conversation Characteristics

Call	Sets
Set_Conversation_Type (cmsct)	Conversation type
Set_Mode_Name (cmsmn)	Mode name
Set_Partner_LU_Name (cmspln)	Partner LU name
Set_TP_Name (cmstpn)	Partner program's TP name
Set_Return_Control (cmsrc)	Return control
Set_Sync_Level (cmssl)	Synchronization level
AIX, LINUX	
Set_Conversation_Context (cmsctx)	Conversation context (groups this conversation with a previous one)
Set_Conversation_Security_Type (cmscst)	Conversation security type
Set_Conversation_Security_User_ID (cmscsu)	Security user ID
Set_Conversation_Security_Password (cmscsp)	Security password

Allocate (cmallc)

This call is issued by the invoking program to allocate a conversation with the partner program, using the current conversation characteristics. The type of conversation allocated depends on the conversation type characteristic (mapped or basic).

Accept_Conversation (cmaccp)

This call is issued by the invoked program to accept the incoming conversation and set certain conversation characteristics. Upon successful execution of this call, CPI-C generates and returns a conversation identifier. Accept_Conversation always operates in blocking mode.

Accept_Incoming (cmacci)

AIX, LINUX

This call is issued by the invoked program to accept an incoming conversation for which it previously issued Initialize_For_Incoming. It is similar to Accept_Conversation, but can operate in nonblocking mode if required (Accept_Conversation always operates in blocking mode).

Sending data

The following calls are used to send data to the partner program.

Set_Send_Type (cmsst)

This call sets the conversation's send type. The send type specifies how data will be sent by the Send_Data call. The Send_Data call can include the function of the Flush, Confirm, Prepare_To_Receive, or Deallocate call (equivalent to issuing Send_Data, followed by the other call), or it can simply send data without performing any other function. The send type value affects all subsequent Send_Data calls. It can be changed by issuing the Set_Send_Type call again.

Send_Data (cmsend)

This call puts data in the local LU's send buffer for transmission to the partner program.

If the send type (specified by the Set_Send_Type call) includes the function of the Flush, Confirm, Prepare_To_Receive, or Deallocate call, the data is transmitted to the partner LU (and partner program) immediately. Otherwise, the data accumulates in the local LU's send buffer, and is sent when one of the following occurs:

- The send buffer fills up
- The local program issues one of the following calls, which flush the LU's send buffer:
  - Flush
  - Confirm
  - Deallocate
  - Prepare_To_Receive
  - Receive (with the receive type set to `CM_RECEIVE_AND_WAIT`)

Flush (cmflus)

This call sends the contents of the local LU's send buffer to the partner LU (and program). If the send buffer is empty, no action takes place.

Confirm (cmcfm)

This call sends the contents of the local LU's send buffer and a confirmation request to the partner program and waits for confirmation.

Request_To_Send (cmrts)

This call notifies the partner program that the local program wants to send data. The partner program can respond to this request by changing to Receive state so that the local program changes to Send state, or can ignore the request.

Receiving Data

The following calls enable a program to receive data from its partner program.

Set_Prepare_To_Receive_Type (cmsptr)

This call sets the conversation's prepare-to-receive type, which specifies whether subsequent Prepare_To_Receive calls will include Flush or Confirm functionality. The prepare-to-receive type affects all subsequent Prepare_To_Receive calls. It can be changed by issuing the Set_Prepare_To_Receive_Type call again.

Prepare_To_Receive (cmptr)

This call changes the state of the conversation for the local program from Send to Receive, making it possible for the local program to begin receiving data. Before changing the conversation state, this call performs the equivalent of the Flush or Confirm call.

Set_Receive_Type (cmsrt)

This call sets the conversation's receive type, which specifies whether a program issuing a Receive call will wait for data to arrive if data is not available. The receive type value affects all subsequent Receive calls. It can be changed by issuing the Set_Receive_Type call again.

Receive (cmrcv)

Issuing this call while the conversation is in Receive state causes the local program to receive any data that is currently available from the partner program. If no data is available and the receive type is set to `CM_RECEIVE_AND_WAIT`, the local program waits for data to arrive. If the receive type is set to `CM_RECEIVE_IMMEDIATE`, the program does not wait.

Issuing this call while the conversation is in Send or Send-Pending state is allowed only if the receive type is set to `CM_RECEIVE_AND_WAIT`. This flushes the LU's send buffer and changes the conversation state to Receive. The local program then begins to receive data.

Set_Fill (cmsf)

This call sets the conversation's fill type, which specifies whether programs will receive data in the form of logical records or as a specified length of data. It only has an effect in basic conversations. The fill value affects all subsequent Receive calls. It can be changed by issuing the Set_Fill call again.

Converting Data Between ASCII and EBCDIC

The following calls enable a program to translate local data from ASCII to EBCDIC before sending it to the partner program, or translate data received from the partner program from EBCDIC to ASCII. The program needs to use these functions only if the partner program requires data to be in EBCDIC.

Convert_Incoming (cmcnvi)

This call converts an EBCDIC data string into ASCII.

Convert_Outgoing (cmcnvo)

This call converts an ASCII data string into EBCDIC.

WINDOWS

The program can also use the CSV CONVERT verb to convert data between ASCII and EBCDIC. Refer to the *Communications Server for AIX CSV Programmer's Guide* for more information.

Confirming Receipt of Data and Reporting Errors

The following calls confirm receipt of data or report an error.

Confirmed (cmcfmd)

This call replies to a confirmation request from the partner program. It informs the partner program that the local program has not detected an error in the received data. Because the program issuing the confirmation request waits for a confirmation, the Confirmed call synchronizes the processing of the two programs.

Set_Error_Direction (cmsed)

This call specifies whether a program detected an error while receiving data or while preparing to send data. Error direction is relevant only when a program issues the Send_Error call in Send-Pending state.

Set_Log_Data (cmsld)

This call specifies a log message (log data) and its length to be sent to the partner LU. This call only has an effect in basic conversations. If present, log data is sent when the Send_Error call is issued or when the conversation is abnormally deallocated. After the log data is sent CPI-C resets the log data to null and the log data length to `0` (zero).

Send_Error (cmserr)

This call notifies the partner program that the local program has encountered an application-level error. The local program can use the Send_Error call for such purposes as informing the partner program of an error encountered in received data, rejecting a confirmation request, or truncating an incomplete logical record it is sending.

Issuing Calls in Nonblocking Mode

AIX, LINUX

This section does not apply to Java CPI-C. Java CPI-C functions always operate in blocking mode; that is, the function does not return control to the application until the requested processing has completed. The functions described in this section are not available in Java CPI-C.

The following calls enable the program to specify that subsequent CPI-C calls can operate in nonblocking mode, to check whether a previous nonblocking call has completed, or to wait for a nonblocking call to complete.

For details on using nonblocking mode, see “AIX or Linux Considerations” on page 37and “Windows Considerations” on page 41. (See also “Cancel_Conversation (cmcanc)” on page 26; this cancels a previous nonblocking call and also deallocates the conversation.)

Set_Processing_Mode (cmspm)

This call sets the conversation’s processing mode to blocking (calls do not return until processing has completed) or nonblocking (calls can return immediately even though processing is not yet complete).

Check_For_Completion (cmchck)

AIX, LINUX

This call checks whether there is an outstanding nonblocking function on any of the program’s conversations for which processing has completed. If there is such a function, it returns the conversation ID of the appropriate conversation; the program then calls Wait_For_Conversation to get the results of the nonblocking function. This call enables the program to check for completion of nonblocking functions without having to suspend (unlike Wait_For_Conversation, which suspends until a function has completed). Check_For_Completion does not return the results of the previous call; the program must use Wait_For_Conversation to do this before it can issue further calls on this conversation.

Wait_For_Conversation (cmwait)

This call waits for processing of a previous nonblocking function to complete. If the program is involved in multiple concurrent conversations, this call acts across all conversations, and returns when a function completes on any of them.

WINDOWS

The Wait_For_Conversation call is supported on Windows systems for compatibility with other Windows CPI-C implementations; however, new Windows applications should use Specify_Windows_Handle (described below) instead of this call.

Specify_Windows_Handle (xchwnd)

This call specifies a Windows handle to which CPI-C posts the results of nonblocking functions. The application receives a message from CPI-C, sent to this Windows handle, when a nonblocking function completes; it does not need to use Wait_For_Conversation to obtain the results of verb completion.

Issuing Calls in Blocking Mode

The following calls enable a Windows program to manage how subsequent CPI-C calls operate in blocking mode. (See also “Set_Processing_Mode (cmspm)”; this specifies whether subsequent calls operate in blocking mode or nonblocking mode.) For more information about blocking calls, see “Blocking Calls” on page 42.

WinCPICIsBlocking
: Checks whether there is a blocking CPI-C call outstanding for this application.

WinCPICSetBlockingHook
Specifies the blocking procedure that CPI-C uses while processing blocking calls; this replaces CPI-C's default blocking procedure. The blocking procedure is called repeatedly until CPI-C has finished processing the call.

WinCPICUnhookBlockingHook
Unregisters the blocking procedure specified by a previous `WinCPICSetBlockingHook` call, so that CPI-C reverts to using the default blocking procedure.

Getting Information

The following calls provide information to programs.

Extract_* Calls

The Extract_* calls, listed in Table 8 retrieve information about the characteristics of a specified conversation.

Table 8. Extract_* Calls and Actions

Call	Retrieves
Extract_Conversation_Security_Type (xcecst)(not available in Java CPI-C)	Security type
Extract_Conversation_State (cmecs)	Conversation state
Extract_Conversation_Type (cmect)	Conversation type
AIX, LINUX	
Extract_Conversation_Context (cmectx)	Conversation context
Extract_Max_Buffer_Size (cmembs)	Maximum size of data buffer used for Send_Data and Receive calls
Extract_Security_User_ID (cmesui)	Security user ID
WINDOWS	
Extract_Conversation_Security_User_ID (cmecsu)	Security user ID
Extract_Mode_Name (cmemn)	Mode name
Extract_Partner_LU_Name (cmepln)	Partner LU name
Extract_TP_Name (cmetpn)	TP name that was specified on the incoming Allocate request
Extract_Sync_Level (cmesl)	Synchronization level

Test_Request_to_Send_Received (cmtrts)

This call determines whether a request-to-send notification has been received from the partner program.

Ending a Conversation

The following calls end a conversation.

Set_Deallocate_Type (cmsdt)

This call specifies how the conversation is to be deallocated. The deallocation instructions specified by this call take effect when the Deallocate call is issued or when the send type is set to CM_SEND_AND_DEALLOCATE and the Send_Data call is issued.

Deallocate (cmdeal)

This call deallocates a conversation between two programs. Before deallocating the conversation, this call performs the equivalent of the Flush or Confirm call, depending on the current conversation synchronization level and deallocate type.

Cancel_Conversation (cmcanc)

This call cancels any incomplete call on a conversation, and deallocates the conversation. (An incomplete call is one that was issued in nonblocking mode and returned CM_OPERATION_INCOMPLETE.)

In Java CPI-C, nonblocking calls are not supported and so there cannot be an incomplete call outstanding. Cancel_Conversation is equivalent to Deallocate except that it does not write log data to the local error log.

WinCPICCleanup

WINDOWS

This call unregisters the application as a Windows CPI-C application, after it has finished issuing CPI-C calls. A Windows CPI-C application must use this call before terminating, and must not issue any other CPI-C calls after it has issued this call.

Administering Side Information

These functions are not available in Java CPI-C.

The calls summarized in Table 9 enable CPI-C applications to add, replace, retrieve, or delete side information entries.

Table 9. Calls to Add, Replace, Retrieve, or Delete Side Information

Call	Action
Set_CPIC_Side_Information (xcmssi)	Add or replace side information entry.
Extract_CPIC_Side_Information (xcmesi)	Retrieve side information entry.
Delete_CPIC_Side_Information (xcmdsi)	Delete side information entry.

Initial Conversation Characteristics

CPI-C maintains a set of internal values, called characteristics, for each conversation. Some characteristics affect the overall operation of the conversation, such as the conversation type. Others affect the operation of specific calls, such as the receive type.

Many of these characteristics are initially derived from the side information stored in the CS/AIX configuration file; see “Side Information” on page 30. The

Initialize_Conversation call specifies the symbolic destination name (the *sym_dest_name* parameter) associated with the desired side information table entry.

Table 10 lists the conversation characteristics, how they are set or changed by the following conversation start-up calls, and which call can change a given value.

- Initialize_Conversation
- Accept_Conversation
- Initialize_For_Incoming
- Accept_Incoming

AIX, LINUX

The calls Initialize_For_Incoming and Accept_Incoming are always used together. A characteristic is normally set by one of these calls and not changed by the other.

WINDOWS

The Initialize_For_Incoming and Accept_Incoming calls are not supported on Windows systems. All references to these calls should be ignored for Windows systems.

For a complete explanation of a characteristic, see the description of the Set_* call associated with it in Chapter 3, "CPI-C Calls," on page 47. For example, the conversation type is described in the section on the Set_Conversation_Type call.

Table 10. Changing Initial Conversation Characteristics

Conversation State	
Initialize_Conversation sets:	`CM_INITIALIZE_STATE`
Accept_Conversation sets:	`CM_RECEIVE_STATE`
Initialize_For_Incoming sets:	`CM_INITIALIZE_INCOMING_STATE`
Accept_Incoming sets:	`CM_RECEIVE_STATE`
Can be changed by:	Many CPI-C calls; see the *State Change* sections at the end of each CPI-C call description in Chapter 3, "CPI-C Calls," on page 47 for information about state changes resulting from the call.
Conversation Type	
Initialize_Conversation sets:	`CM_MAPPED_CONVERSATION`
Accept_Conversation sets:	The value specified by the invoking program.
Initialize_For_Incoming sets:	(Not set)
Accept_Incoming sets:	The value specified by the invoking program.
Can be changed by:	Set_Conversation_Type
Deallocate Type	
Initialize_Conversation sets:	`CM_DEALLOCATE_SYNC_LEVEL`
Accept_Conversation sets:	`CM_DEALLOCATE_SYNC_LEVEL`
Initialize_For_Incoming sets:	`CM_DEALLOCATE_SYNC_LEVEL`

Table 10. Changing Initial Conversation Characteristics (continued)

Accept_Incoming sets:	(Not changed)
Can be changed by:	Set_Deallocate_Type
Error Direction	
Initialize_Conversation sets:	`CM_RECEIVE_ERROR`
Accept_Conversation sets:	`CM_RECEIVE_ERROR`
Initialize_For_Incoming sets:	`CM_RECEIVE_ERROR`
Accept_Incoming sets:	(Not changed)
Can be changed by:	Set_Error_Direction
Fill	
Initialize_Conversation sets:	`CM_FILL_LL`
Accept_Conversation sets:	`CM_FILL_LL`
Initialize_For_Incoming sets:	`CM_FILL_LL`
Accept_Incoming sets:	(Not changed)
Can be changed by:	Set_Fill
Log Data	
Initialize_Conversation sets:	Null string
Accept_Conversation sets:	Null string
Initialize_For_Incoming sets:	Null string
Accept_Incoming sets:	(Not changed)
Can be changed by:	Set_Log_Data
Local LU Name	
Initialize_Conversation sets:	The local LU alias from one of a number of different sources (see "Specifying the Local LU" on page 34).
Accept_Conversation sets:	The LU alias for the session the conversation start-up request arrived on.
Initialize_For_Incoming sets:	(Not set)
Accept_Incoming sets:	The LU alias for the session the conversation start-up request arrived on.
Can be changed by:	Set_Local_LU_Name
Mode Name	
Initialize_Conversation sets:	The mode name from the side information, or a null string if no *sym_dest_name* is specified.
Accept_Conversation sets:	The mode name for the session the conversation start-up request arrived on.
Initialize_For_Incoming sets:	(Not set)
Accept_Incoming sets:	The mode name for the session the conversation start-up request arrived on.
Can be changed by:	Set_Mode_Name
Partner LU Name	
Initialize_Conversation sets:	The partner LU name from the side information, or a single blank if no *sym_dest_name* is specified.
Accept_Conversation sets:	The partner LU name for the session the conversation start-up request arrived on.
Initialize_For_Incoming sets:	(Not set)
Accept_Incoming sets:	The partner LU name for the session the conversation start-up request arrived on.

Table 10. Changing Initial Conversation Characteristics (continued)

Can be changed by:	Set_Partner_LU_Name
Prepare-to-Receive Type	
Initialize_Conversation sets:	CM_PREP_TO_RECEIVE_SYNC_LEVEL
Accept_Conversation sets:	CM_PREP_TO_RECEIVE_SYNC_LEVEL
Initialize_For_Incoming sets:	CM_PREP_TO_RECEIVE_SYNC_LEVEL
Accept_Incoming sets:	(Not changed)
Can be changed by:	Set_Prepare_To_Receive_Type
Processing Mode (Blocking or Nonblocking)	
Initialize_Conversation sets:	CM_BLOCKING
Accept_Conversation sets:	CM_BLOCKING
Initialize_For_Incoming sets:	CM_BLOCKING
Accept_Incoming sets:	(Not changed)
Can be changed by:	Set_Processing_Mode
Receive Type	
Initialize_Conversation sets:	CM_RECEIVE_AND_WAIT
Accept_Conversation sets:	CM_RECEIVE_AND_WAIT
Initialize_For_Incoming sets:	CM_RECEIVE_AND_WAIT
Accept_Incoming sets:	(Not changed)
Can be changed by:	Set_Receive_Type
Return Control	
Initialize_Conversation sets:	CM_WHEN_SESSION_ALLOCATED
Accept_Conversation sets:	(Not applicable)
Initialize_For_Incoming sets:	(Not applicable)
Accept_Incoming sets:	(Not applicable)
Can be changed by:	Set_Return_Control
Security Password	
Initialize_Conversation sets:	The password contained in the side information, or a single blank if no *sym_dest_name* is specified.
Accept_Conversation sets:	(Not applicable)
Initialize_For_Incoming sets:	(Not applicable)
Accept_Incoming sets:	(Not applicable)
Can be changed by:	Set_Conversation_Security_Password
Security Type	
Initialize_Conversation sets:	The security type contained in the side information, or CM_SECURITY_SAME if no *sym_dest_name* is specified.
Accept_Conversation sets:	(Not applicable)
Initialize_For_Incoming sets:	(Not applicable)
Accept_Incoming sets:	(Not applicable)
Can be changed by:	Set_Conversation_Security_Type
Security User ID	
Initialize_Conversation sets:	The user ID contained in the side information, or a single blank if no *sym_dest_name* is specified.
Accept_Conversation sets:	The value specified by the invoking program.
Initialize_For_Incoming sets:	(Not set)

Table 10. Changing Initial Conversation Characteristics (continued)

Accept_Incoming sets:	The value specified by the invoking program.
Can be changed by:	Set_Conversation_Security_User_ID
Send Type	
Initialize_Conversation sets:	`CM_BUFFER_DATA`
Accept_Conversation sets:	`CM_BUFFER_DATA`
Initialize_For_Incoming sets:	`CM_BUFFER_DATA`
Accept_Incoming sets:	(Not changed)
Can be changed by:	Set_Send_Type
Synchronization Level	
Initialize_Conversation sets:	`CM_NONE`
Accept_Conversation sets:	The value specified by the invoking program.
Initialize_For_Incoming sets:	(Not set)
Accept_Incoming sets:	The value specified by the invoking program.
Can be changed by:	Set_Sync_Level
TP Name of the Invoked Program (As Seen by the Invoking Program)	
Initialize_Conversation sets:	The TP name contained in the side information, or a single blank if no *sym_dest_name* is specified.
Accept_Conversation sets:	(Not applicable)
Initialize_For_Incoming sets:	(Not applicable)
Accept_Incoming sets:	(Not applicable)
Can be changed by:	Set_TP_Name
TP Name of the Invoked Program (As Seen by the Invoked Program)	
Initialize_Conversation sets:	(Not applicable)
Accept_Conversation sets:	The value specified by the invoking program.
Initialize_For_Incoming sets:	(Not set)
Accept_Incoming sets:	The value specified by the invoking program.
Can be changed by:	Specify_Local_TP_Name (to indicate one or more names for which to accept incoming allocates)

Side Information

The information required for two programs to communicate is stored in CPI-C side information entries in the CS/AIX configuration file. You will need to coordinate with your System Administrator to ensure that it contains what you need. For additional information about configuration, refer to the *Communications Server for AIX Administration Guide*.

Each side information entry is identified by a Symbolic Destination Name, which is the *sym_dest_name* parameter specified by the Initialize_Conversation call. The parameter *sym_dest_name* is an 8-byte ASCII character string and can contain any displayable characters.

If you are developing commercial programs or programs that will be installed on multiple machines within your organization, you may want to include logic to use a different *sym_dest_name* for each copy of the program.

Each side information entry contains the following fields:

- Local LU alias
- Partner LU name
- Partner program type and name
- Mode name
- Conversation security type
- Security user ID and password
- Application-specified side information

Local LU Alias

This is the alias of the local LU to be used to allocate conversations. It consists of up to eight ASCII characters. For the allowed characters, see "Set_Local_LU_Name (cmslln)" on page 131.

Partner LU Name

This is the name by which the partner LU is known to the local program. It can be an alias of up to eight ASCII characters or a fully qualified network name of up to 17 characters. For the allowed characters, see "Set_Partner_LU_Name (cmspln)" on page 136.

Partner Program Type and Name

These fields indicate whether the partner program is an application program or SNA service program, and the partner program name. An application program name can contain up to 64 ASCII characters. A service program can contain up to four characters. For the allowed characters, see "Set_TP_Name (cmstpn)" on page 147.

Mode Name

This name represents a set of characteristics to be used in an LU-to-LU session. The mode name can contain up to eight ASCII characters. For the allowed characters, see "Set_Mode_Name (cmsmn)" on page 134.

Conversation Security Type

This field indicates whether security will be used and if so, what type. The security type can specify that CPI-C must send a user ID and password when allocating a conversation with the invoked program. For an invoked program that in turn invokes another program, the security type can inform the second invoked program that security has already been verified.

For further information about conversation security, see "Set_Conversation_Security_Type (cmscst)" on page 116.

Security User ID and Password

If the remote program uses conversation security, and does not accept an "already verified" indication, a valid combination of user ID and password is required to access the invoked program. The user ID and password can be up to 10 ASCII characters. For the allowed characters, see "Set_Conversation_Security_User_ID (cmscsu)" on page 118 and "Set_Conversation_Security_Password (cmscsp)" on page 114.

Application-Specified Side Information

AIX, LINUX

Note: The functions described in this section are not available in Java CPI-C. A Java CPI-C application cannot maintain its own CPI-C side information entries. However, it can override individual parameters in the side information, or determine their values, by using Set_* or Extract_* functions for each required parameter.

An application can override the side information stored in the configuration file to maintain its own side information entries, using the following calls:

- Set_CPIC_Side_Information (to define a side information entry associated with a specified *sym_dest_name*; if the *sym_dest_name* is already defined in the configuration file, the new information overrides the configuration file)
- Delete_CPIC_Side_Information (to indicate that an entry defined by the application, or one defined in the configuration file, is no longer available for use by this application)
- Extract_CPIC_Side_Information (to return the contents of a side information entry—either an entry defined by the application, or one defined in the configuration file)

The modified information then applies only to this application; it does not affect other applications, and does not change the configuration file. The modified information is discarded when the application ends.

These calls are not part of IBM CPI-C 2.0; they are provided for compatibility with X/Open CPI-C. In addition, in the side information structure used by these calls, the user ID and password parameters are defined as eight characters (as in X/Open CPI-C) instead of 10 (as in IBM CPI-C 2.0). This leads to the following restrictions:

- If the partner application requires a user ID or password of more than eight characters, you cannot specify it using Set_CPIC_Side_Information. You must either use a side information entry defined in the configuration file, or define one using Set_CPIC_Side_Information and then override the user ID or password using the Set_Conversation_Security_User_ID or Set_Conversation_Security_Password call.
- If the side information entry in the configuration file contains a user ID of more than eight characters, you cannot extract it using Extract_CPIC_Side_Information. You must use the Extract_Security_User_ID call. (This does not apply to the password, because CPI-C does not allow the application to extract it.)

Configuration

The following are considerations when configuring CS/AIX:

- In addition to maintaining the side information (specified by *sym_dest_name*), the System Administrator must define the following entities during configuration to enable CPI-C applications to use CS/AIX's LU 6.2 services:
  - Modes

  - Local LUs
  - Partner LUs
  - Invokable TPs
  - Security user IDs and passwords

  For further information, refer to the *Communications Server for AIX Administration Guide*.
- If you want to enable autostart sessions, set the *auto_act* parameter on the mode. For more information about defining modes, refer to *Communications Server for AIX Administration Guide*.

Specifying the Local TP Name

When a program issues the Initialize_Conversation, Initialize_Conversation_For_Incoming, or Accept_Conversation call, the CPI-C library generates an instance of a transaction program (TP). You can specify the name of this TP in a number of different ways, described below.

The methods are listed in order of precedence. This means that, if you specify a name using the first method, the CPI-C library uses this name and ignores any name that you specify using the second or later methods. If you do not use the first method but specify a name using the second method, the CPI-C library uses this name and ignores any name that you specify using the third or later methods, and so on.

- For invoking programs, the TP name is only used as an identifier in log and trace files.
- For operator-started invoked programs, the TP name must be set correctly because the value is used to route inbound allocation requests to the appropriate program. The Accept_Conversation, or Accept_Incoming call from the invoked program completes when an inbound allocation request arrives for this TP name.
- For automatically-started invoked programs, the TP name need not be specified because it is taken from the inbound allocation request.

Note: The local TP name is distinct from the partner TP name set in the Set_TP_Name call.

Specify_Local_TP_Name

The program can use this call to specify the TP name.

Context

If there is another TP from which the context is copied, the TP name is taken from that other TP. For more information about context, see “Multiple Conversations” on page 12.

APPCTPN Environment Variable

The TP name can be specified using the `APPCTPN` environment variable.

AIX, LINUX

On AIX or Linux systems the TP name is specified in the `APPCTPN` environment variable. This environment variable can be set in the following ways:

- The program can issue a `putenv` call

- You can set it in the AIX or Linux shell. For example, in the Korn shell you would issue the following command:

  export APPCTPN=MYTP
- If you are using automatically started invoked TPs, you can set it using the environment field of the CS/AIX invokable TP data file.

WINDOWS

On Windows systems the TP name can be specified either using the APPCTPN environment variable, or in the registry. CPI-C checks the environment variable first, and uses this name if it is specified; it uses the registry entry only if the environment variable is not specified. You may need to use environment variables if you are using Windows Terminal Server and need to run multiple copies of the same application using different local LUs.

The registry key is

\\HKEY_LOCAL_MACHINE\SOFTWARE\SNA Client\SxClient\Parameters\MyExeName

where MyExeName is the file name of the program, without the **.exe** extension.

The APPCTPN value under this registry key specifies the TP name.

Default Value

If the TP name is not set by any of the methods described in the previous sections then it is set to the default value CPIC_DEFAULT_TPNAME.

Specifying the Local LU

The local LU that an invoking CPI-C TP uses can be specified in a number of ways which are described below.

Note: The local LU for an invoked TP is not specified like this, but is defined by the partner LU value specified in the allocate request.

If the LU specified is a dependent LU, multiple concurrent conversations are not supported (because dependent LUs cannot support multiple sessions).

The different ways that you can set the local LU alias are described in the following sections. The methods are listed in order of precedence. This means that, if you specify a local LU alias using the first method, the CPI-C library uses this name and ignores any alias that you specify using the second or later methods. If you do not use the first method but specify a local LU alias using the second method, the CPI-C library uses this alias and ignores any alias that you specify using the third or later methods, and so on.

Set_Local_LU_Name

The program can issue this call to specify the local LU alias after the Initialize_Conversation call has completed. This call only affects the TP from which it is issued. It does not modify the side information stored in the configuration file.

Note: This call is not part of the standard CPI-C specification, and may not be available in other implementations. You may want to avoid using this function, or to restrict it to a few specific routines which can be modified easily, if you need to ensure that your application can be used with other CPI-C implementations.

Context

If there is another TP from which the context is copied, the local LU name is taken from that other TP. For more information about context, see "Multiple Conversations" on page 12.

APPCLLU Environment Variable

The local LU alias can be specified using the `APPCLLU` environment variable.

AIX, LINUX

On AIX or Linux systems this environment variable can be set in the following ways:

- The program can issue a `putenv` call
- You can set it in the AIX or Linux shell. For example, in the Korn shell you would issue the following command:

  export APPCLLU=MYLU

WINDOWS

On Windows systems the local LU alias can be specified either using the `APPCLLU` environment variable, or in the registry. CPI-C checks the environment variable first, and uses this alias if it is specified; it uses the registry entry only if the environment variable is not specified. You may need to use environment variables if you are using Windows Terminal Server and need to run multiple copies of the same application using different local LUs.

The registry key is

\\HKEY_LOCAL_MACHINE\SOFTWARE\SNA Client\SxClient\Parameters\MyExeName

where MyExeName is the file name of the program, less the **.exe** extension.

The `APPCLLU` value under this registry key specifies the local LU alias.

Side Information

The local LU alias is part of the side information configured for each symbolic destination name. TPs select which of these to use in the Initialize_Conversation call.

Note: Programs can modify the side information. For more information, see "Administering Side Information" on page 26.

Default Local LU

Local LUs can be configured to be a part of the default pool of APPC LUs. If no other local LU alias is specified, any suitable LU from this pool is used.

Control Point LU

CS/AIX normally has one control point (CP) LU defined on each node. If no other local LU alias is defined then the CP LU is used.

How Programs Get Started

A conversation occurs between an invoking program and an invoked program. The invoking program is started by a user entering a command or by a batch command. The invoked program can either be started manually by a user or automatically by CS/AIX.

Invoked Program: Automatically Started

An invoked program can be configured to start automatically under one of the following conditions:

- The first time an inbound allocation request is received by the LU that serves the invoked program. A program started in this manner is called a queued, automatically started program (or queued auto-started TP).

  If the invoked program is not running, the first inbound allocation request starts it; a response to the allocate request is held until the Accept_Conversation or Accept_Incoming call in the invoked program is executed.

  If the invoked program is already running, the inbound allocation request waits until the invoked program issues another Accept_Conversation or Accept_Incoming call, or until it finishes running and can be restarted.
- Each time an inbound allocation request is received by the LU that serves the invoked program, a new instance of the program is loaded and started. A program started in this manner is called a nonqueued, automatically started program.

In general, the inbound allocation request waits until the invoked program is started and issues an Accept_Conversation or Accept_Incoming call. However, the definition of the invoked program's local LU includes a timeout value, so that the inbound allocation request fails if the timeout is reached before the invoked program issues an Accept_Conversation or Accept_Incoming call.

The definition of the invoked TP (in the CS/AIX invokable TP data file) includes a second timeout value, which determines how long an Accept_Conversation or Accept_Incoming call waits for an inbound allocation request. The call fails if this timeout is reached before an inbound allocation request is received. This timeout value does not apply to a nonqueued program, because the program is always started in response to an inbound allocation request and so there is always one pending.

Invoked Program: User-Started

If an invoked program is configured to be started by a user, the user can start the invoked program either before or after the invoking program. A program started in this manner is called a queued, operator-started program.

If the user starts the invoking program before starting the invoked program, the inbound allocation request to the invoked program waits until the invoked

program is started and issues an Accept_Conversation or Accept_Incoming call. However, the definition of the invoked program's local LU includes a timeout value, so that the inbound allocation request fails if the timeout is reached before the invoked program is started and issues an Accept_Conversation or Accept_Incoming call.

If the user starts the invoked program before the invoking program issues the Allocate call, the Accept_Conversation or Accept_Incoming call issued by the invoked program waits for an inbound allocation request. The definition of the invoked TP (in the CS/AIX invokable TP data file) includes a second timeout value, which determines how long an Accept_Conversation or Accept_Incoming call waits for an inbound allocation request. The call fails if this timeout is reached before an inbound allocation request is received.

AIX or Linux Considerations

AIX, LINUX

This section summarizes the information you need to consider when writing CPI-C applications for AIX or Linux systems.

If you are writing Java CPI-C applications, see "Java CPI-C Considerations" on page 38.

CPI-C Header File

The header file to be used with CPI-C applications is **cmc.h**. This file contains the definitions of all CPI-C entry points. It also includes the common interface header file **values_c.h**; these two files contain all the constants defined for supplied and returned parameter values at the CPI-C interface. Both files are stored in **/usr/include/sna** (AIX) or **/opt/ibm/sna/include** (Linux).

Multiple Processes

If the process that started the conversation forks to create a child process, the child process cannot use the `conversation_ID` that was returned to the parent process. It can, however, issue its own Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call to obtain its own `conversation_ID`.

Two or more instances of the same program can run as different processes, but each instance will be assigned its own `conversation_ID`.

You can write an application in which one process contains many conversations, each with its own `conversation_ID`. However, you need to design the application carefully to avoid "deadlock" situations, in which a CPI-C call is unable to complete because of the state of other conversations in the same process. This might happen if the program is waiting on one conversation for information to be sent to it before returning some other data, and another conversation from the same process is waiting for this data before it can send the information originally required by the first conversation. To some extent this can be avoided by using a separate process for each conversation.

Compiling and Linking the CPI-C Application

AIX Applications

To compile and link 32–bit applications, use the following options:

-bimport:/usr/lib/sna/cpic_r.exp -I /usr/include/sna

To compile and link 64–bit applications, use the following options:

-bimport:/usr/lib/sna/cpic_r64_5.exp -I /usr/include/sna

Linux Applications

Before compiling and linking a CPI-C application, specify the directory where shared libraries are stored, so that the application can find them at run time. To do this, set the environment variable `LD_RUN_PATH` to **/opt/ibm/sna/lib**, or to **/opt/ibm/sna/lib64** if you are compiling a 64–bit application.

To compile and link 32–bit applications, use the following options:

-I /opt/ibm/sna/include -L /opt/ibm/sna/lib -lcpic -lappc -lsna_r -lpthread

To compile and link 64–bit applications, use the following options:

-I /opt/ibm/sna/include -L /opt/ibm/sna/lib64 -lcpic -lappc -lsna_r -lpthread

Java CPI-C Considerations

This section summarizes the information you need to consider when writing Java CPI-C applications.

Using Java CPI-C Classes

The Java CPI-C package is named **COM.ibm.eNetwork.cpic**. This package consists of a Java class that contains:

- A method for each of the supported CPI-C calls
- Classes for use as parameters to these calls

When writing a Java program to use the CPIC class, use the following import statement in the Java source to import the CPIC package:

import COM.ibm.eNetwork.cpic.*;

Constant Values

The Java CPI-C class defines a number of constant values for the maximum length in bytes of specific CPI-C parameters. These constants are shown in Table 11. You should use these constants in your program rather than specifying the lengths explicitly.

Table 11. Java CPI-C Constants

Parameter Length	Java CPI-C Constant
Conversation ID Length	CM_CID_SIZE

Table 11. Java CPI-C Constants (continued)

Parameter Length	Java CPI-C Constant
Context ID Length	CM_CTX_SIZE
Log Data Size	CM_LD_SIZE
Mode Name Length	CM_MN_SIZE
Partner LU Name Length	CM_PLN_SIZE
Security Password Length	CM_PW_SIZE
Security User ID Length	CM_UID_SIZE
Symbolic Destination Name Length	CM_SDN_SIZE
Transaction Program (TP) Name length	CM_TPN_SIZE

Parameter Type Classes

Many parameters used in CPI-C functions take one of a set of two or more defined values. In the Java CPI-C package, each of these parameter types is defined as a class containing the valid values. For example, the CPICSyncLevel class is used in the functions Set_Sync_Level (cmssl) and Extract_Sync_Level (cmesl), and can take a value of either CM_NONE or CM_CONFIRM.

The description of each CPI-C function in Chapter 3, "CPI-C Calls," on page 47 gives the appropriate CPI-C parameter class type and the valid values. For example, in Set_Sync_Level (cmssl), the *sync_level* parameter is listed as being of type CPICSyncLevel, and the description of parameters for this function lists the valid values as CM_NONE or CM_CONFIRM.

Because the constant values associated with a Java class are defined in the class, you must access them by referring to the class as well as the specific value. For example, to specify no confirm synchronization, you must set the *sync_level* parameter of the Set_Sync_Level function to `CPICSyncLevel.CM_NONE`.

Each of these classes has the following methods in addition to the constructor:

int intValue()
: Returns the value stored in the object.

int intValue(*int_value***)**
: Sets the value stored in the object to the supplied integer value *int_value*, and returns the same value.

  You can also set the value stored in an object during construction of the object, by passing the value in as a parameter to the constructor.

boolean equals(*int_value***)**
: Returns `true` if the value stored in the object is equal to the supplied integer value *int_value*.

boolean equals(*supplied_object***)**
: Returns `true` if the value stored in the object is equal to the value stored in the supplied parameter *supplied_object*. *supplied_object* must itself be an instance of one of the Java CPI-C parameter classes.

The class `CPICReturnCode` has the following additional method:

boolean isOK()
: The application should call this method to determine whether the value stored in a `CPICReturnCode` object is `CM_OK`. The class generates an exception if the stored value is not `CM_OK`.

Usage Example

The following example illustrates how to set up your Java program to use the Java CPI-C class, and how to make an individual CPI-C call.

To import the Java CPI-C package, include the following at the start of your program's source code:

```
import COM.ibm.eNetwork.cpic.*;
```

To use Java CPI-C in your program, create an instance of the Java CPI-C class:

```
CPIC cpicObject = new CPIC();
```

The following steps illustrate how to make the call to each Java CPI-C function, using the Initialize_Conversation (cminit) function as an example.

1. Create and initialize the parameters for the function:

   ```
   byte[] bConversationId = new byte[cpicObject.CM_CID_SIZE];
   String sSymbolicDestination = "testprog";
   CPICReturnCode cpicReturn = new CPICReturnCode(0);
   ```

   Note the use of the constant `CM_CID_SIZE` to set the size of the byte array for the conversation ID, and the use of the CPICReturnCode class to set the initial value of this parameter to zero. The last line of this example could also be split into two lines as follows:

   ```
   CPICReturnCode cpicReturn = new CPICReturnCode();
   cpicReturn.intValue(0);
   ```

2. Issue the function call:

   ```
   cpicObject.cminit(bConversationId,
                     sSymbolicDestination,
                     cpicReturn);
   ```

3. Test the return code against a specific value:

   ```
   if (cpicReturn.intValue() != CPICReturnCode.CM_PARAMETER_ERROR)
   . . .
   ```

   Alternatively, check whether the return code is `CM_OK`:

   ```
   try
   {
         cpicReturn.isOK();
   }
   catch(CPICReturncode c)
   {
   . . . // cpicReturn is not set to CM_OK
   }
   ```

Compiling and Linking the Java CPI-C Application

Before compiling and linking a Java CPI-C application, specify the directory where Java classes are stored. To do this, set and export the environment variable `CLASSPATH` to **/usr/lib/sna/java/cpic.jar:.** (AIX) or **/opt/ibm/sna/java/cpic.jar:.** (Linux).

Compile and link the application using the Java compiler **javac** in the normal way.

Running the Java CPI-C Application

Before running a Java CPI-C application, you need to specify the directory where libraries are stored, so that the application can find them at run time.

To do this, set and export the appropriate environment variable as follows.

export LD_LIBRARY_PATH=/usr/lib/sna

You may also need to set and export the `APPCTPN` environment variable to specify the local TP name for the application, as described in "Specifying the Local TP Name" on page 33.

Run the application using the Java interpreter **java** in the normal way.

Windows Considerations

WINDOWS

This section summarizes processing considerations you need to be aware of when developing programs on a Remote API Client on Windows.

Windows CPI-C Files

The header file to be used with Windows CPI-C applications is **wincpic.h**, which contains the definitions of all CPI-C entry points, and the defined constants for supplied and returned parameter values at the Windows CPI-C interface. This file is installed in the subdirectory **/sdk** within the directory where you installed the Remote API Client on Windows software.

The library used to link Windows CPI-C applications is **wcpic32.lib**.

Function Prototypes

The function prototypes for CPI-C calls shown in Chapter 3, "CPI-C Calls," on page 47 are in the format used for AIX or Linux systems. For Windows systems, replace "void *functionname*" with "void WINAPI *functionname*" for each call.

Multiple Processes and Multiple Conversations

Multiple processes cannot have the same conversation identifier. Only the process that issues the Initialize_Conversation or Accept_Conversation call can use the conversation ID returned by the call. Another process wanting to use CPI-C must issue an Initialize_Conversation or Accept_Conversation call to obtain its own conversation ID.

One program can engage in up to 64 simultaneous conversations.

Windows Function Calls

In addition to the standard CPI-C function calls, and the Windows-specific CPI-C function call Specify_Windows_Handle, a Windows application also uses the following functions:

WinCPICStartup
Registers the application as a Windows CPI-C user, and determines whether the CPI-C software supports the level of function required by the application.

WinCPICCleanup
Unregisters the application when it has finished using CPI-C.

WinCPICIsBlocking
Checks whether there is a blocking call outstanding for this application. For more information about the circumstances in which this call may be required, see "Blocking Calls."

WinCPICSetBlockingHook
Specifies the blocking procedure that CPI-C uses while processing blocking calls; this replaces CPI-C's default blocking procedure. The blocking procedure is called repeatedly until processing for the blocking call has completed. For more information, see "Blocking Calls."

WinCPICUnhookBlockingHook
Unregisters the blocking procedure specified by a previous `WinCPICSetBlockingHook` call, so that CPI-C reverts to using the default blocking procedure.

WinCPICExtractEvent
Provides a method for an application to determine the Win32 event handle being used for a CPI-C conversation.

WinCPICSetEvent
Associates a Win32 event handle with verb completion for a CPI-C conversation.

The application must call `WinCPICStartup` before attempting to issue any CPI-C calls.

"Blocking Calls" provides more information about how blocking calls operate in the Windows environment, and how the application should use the `WinCPICIsBlocking`, `WinCPICSetBlockingHook`, and `WinCPICUnhookBlockingHook` calls.

When the application has finished issuing CPI-C calls, it must call `WinCPICCleanup` before terminating; it must not attempt to issue any more CPI-C calls after calling `WinCPICCleanup`.

The Windows function calls are described at the end of Chapter 3, "CPI-C Calls," on page 47.

Blocking Calls

This section describes how blocking CPI-C calls (calls issued with the conversation's processing mode set to `CM_BLOCKING`) operate in the Win32 environment if the calling application is single-threaded. (Typically, a Win32 application would use multiple threads to avoid the problem of a blocking verb blocking the entire application.)

The section also provides information that you need to be aware of when writing applications to use blocking calls.

The Remote API Client provides support for blocking calls on Windows systems to assist in migrating applications from other operating system environments.

However, the use of blocking calls in the Windows environment is strongly discouraged. If you are writing a new application specifically for Windows, you should do the following:

- Use the Specify_Windows_Handle function to specify a Windows handle to which CPI-C posts the results of call completion
- Issue all CPI-C calls in nonblocking mode

Although a blocking call appears to suspend the application until CPI-C has finished processing the call, the CPI-C library has to yield control of the system while waiting for CS/AIX to complete the processing, in order to enable other processes to run. To do this, it uses a "blocking function", which is called repeatedly while the library is waiting; the function enables Windows messages to be sent to other processes. For more information about this function, see "Default Blocking Function."

It is possible for the blocking function to send a message to the application that issued the original blocking call; in this case, the application can be reentered even though it has a blocking call outstanding. In these circumstances, the application can continue with other processing not related to issuing CPI-C calls. However, it cannot issue another blocking call while the first call is outstanding.

The application can check whether a blocking call is outstanding (that is, whether it has been reentered as a result of a received message while the call was outstanding) by using the `WinCPICIsBlocking` function, described in Chapter 3, "CPI-C Calls," on page 47. If this function indicates that a blocking call is outstanding, the application should not attempt to issue further blocking CPI-C calls. It can, however, do the following:

- Continue with other processing
- Issue CPI-C calls on other conversations for which the processing mode is `CM_NON_BLOCKING`

Default Blocking Function

The standard blocking function used by the Windows CPI-C library is as follows:

```
BOOL DefaultBlockingHook (void) {
       MSG msg;
       /* get the next message if any */
       if ( PeekMessage (&msg,0,0,PM_NOREMOVE) ) {
             if ( msg.message == WM_QUIT )
                    return FALSE; // let app process WM_QUIT
       PeekMessage (&msg,0,0,PM_REMOVE);
       TranslateMessage (&msg);
       DispatchMessage (&msg);
}
/* TRUE if no WM_QUIT received */
return TRUE;
}
```

If the application needs to have other processing performed as part of the blocking function, it can specify its own blocking function to replace the default one provided by CPI-C. To do this, it uses the `WinCPICSetBlockingHook` call, described in Chapter 3, "CPI-C Calls," on page 47.

A blocking function must return `FALSE` if it receives a WM_QUIT message; this means that CPI-C returns control to the application, which can then process the message and terminate. Otherwise, the function must return `TRUE`.

Terminating Applications

CPI-C cannot tell when an application terminates under Windows. Therefore if an application must close (for example, if it receives a WM_CLOSE message), the application should issue the `WinCPICCleanup` call. Failure to issue the call leaves the system in an indeterminate state; however, as much cleanup as possible is done when CPI-C later detects that the application has terminated.

Compiling and Linking CPI-C Applications

This section provides information about compiling and linking CPI-C applications on Windows systems.

Compiler Options for Structure Packing

The structures supplied and returned on some CPI-C calls are not packed. Do not use compiler options that change this packing method. BYTE parameters are on BYTE boundaries, WORD parameters are on WORD boundaries, and DWORD parameters are on DWORD boundaries.

Header Files

The header file to be included in Windows CPI-C applications is named **wincpic.h**. This file is installed in the subdirectory **/sdk** within the directory where you installed the Windows Client software.

Load-time linking

To link the application to CPI-C at load time, link the application to the library **wincpic32.lib**.

Run-time linking

To link the application to CPI-C at run time, include the following calls in the application:

- `LoadLibrary` to load the CPI-C dynamic link library **wincpic32.dll**
- `GetProcAddress` to specify `WinCPIC` as the entry point to the dynamic link library
- `FreeLibrary` when the library is no longer required

Writing Portable Applications

The following guidelines are provided for writing CPI-C applications that they are portable to other operating system environments or other CPI-C implementations:

- Include the CPI-C header file without any pathname prefix. Use include options on the compiler to locate the file (see the appropriate section for your operating system, earlier in this chapter). This enables the application to be used in an environment with a different file system.
- Use the symbolic constant names for parameter values and return codes, not the numeric values shown in the header file; this ensures that the correct value will be used regardless of the way these values are stored in memory.
- Include a check for return codes other than those applicable to your current operating system (for example using a “default” case in a switch statement), and provide appropriate diagnostics.
- Some of the CPI-C functions provided by CS/AIX are extensions included for compatibility with X/Open CPI-C, or are not part of the standard CPI-C specification, and may not be available in other implementations. Each of these

extension functions is identified by notes in the introduction to the function description in Chapter 3, "CPI-C Calls," on page 47.

- The X/Open functions are included to allow you to use existing applications written for X/Open CPI-C with CS/AIX. You should not use these functions when writing new applications.
- If you use the extension functions in your application, you may need to rewrite sections of the application for use in other environments. You may want to restrict the use of these functions to a few specific routines, to allow easier modification.

AIX, LINUX

The following guidelines apply to Java CPI-C applications:

- The three functions Extract_Conversation_Context, Set_Conversation_Context, and Set_Local_LU_Name are not part of the standard CPI-C specification, and are not supported by IBM's Java CPI-C for CS/Windows. If you use these functions in your Java CPI-C application, you may need to rewrite sections of the application for use in other Java CPI-C environments. You may want to restrict the use of these functions to a few specific routines, to allow easier modification.
- The Java CPI-C class includes some CPI-C functions not described in this manual, which are defined as part of the Java class but not supported. If you use these unsupported functions in your application, it may compile successfully, but the functions will return an error return code (`CM_CALL_NOT_SUPPORTED`) if the application attempts to use them.

Chapter 3. CPI-C Calls

This chapter describes the CPI-C function calls and the additional Windows-specific function calls used by CPI-C applications. The following information is included:

- An explanation of the information provided for the calls
- The call descriptions

Information Provided for CPI-C Calls

The following information is supplied for each CPI-C call described in this chapter:

- The pseudonym for the call, followed by the actual C function name in parentheses (this information is in the section heading).
- The function prototype for the call, including the parameters used by the call and the data type for each parameter. The prototype of each function is declared in the file **cmc.h** (AIX or Linux systems) or **wincpic.h** (Windows systems).

  WINDOWS

  The function prototypes for CPI-C calls shown in Chapter 3, "CPI-C Calls" are in the format used for AIX or Linux systems. For Windows systems, replace "void *functionname*" with "void WINAPI *functionname*" for each call.

  

- The Java method definition for the CPI-C function, if it is supported in Java CPI-C.
- A description of each supplied and returned parameter. The parameter names are pseudonyms. The actual variable names for these parameters are declared by the application program. The description includes the possible values of the parameter.
- The conversation state or states in which the call can be issued.
- The state or states into which the conversation can change upon return from the call. Conditions that do not cause a state change, such as parameter checks and state checks, are not noted.
- Additional information describing the use of the call.

Data Types

For information on data types in Java CPI-C applications, see "Java CPI-C Considerations" on page 38.

To improve the portability of CPI-C applications, the data types for the parameters supplied to, and received from, CPI-C are established as symbolic constants by #define statements in the CPI-C header file. For example, CM_INT32 represents a 32-bit integer type; CM_PTR represents a pointer type.

This chapter uses these symbolic constants to identify the data types for supplied and returned parameters. When writing applications, you are advised to use these symbolic constants rather than the actual data types.

Data Structures

This section does not apply to Java CPI-C applications, because none of the CPI-C functions supported in Java CPI-C use data structures.

For some CPI-C calls, the application supplies a data structure in which CS/AIX can fill in parameters to return to the application. These data structures may contain parameters marked as "reserved"; some of these reserved parameters are used internally by the CS/AIX software, and others are not used in this version but may be used in future versions. Your application must not attempt to access any of these reserved parameters; instead, it must set the entire contents of the data structure to zero to ensure that all of these parameters are zero, before it sets other parameters that are used by the verb. This ensures that CS/AIX will not misinterpret any of its internally-used parameters, and also that your application will continue to work with future CS/AIX versions in which these parameters may be used to provide new functions.

To set the data structure contents to zero, use `memset`:

memset(my_struct, 0, sizeof(my_struct));

Symbolic Constants

For information on symbolic constant values in Java CPI-C applications, see "Java CPI-C Considerations" on page 38.

Most parameters supplied to and returned by CPI-C are 32-bit integers. To simplify coding, the values for these parameters are represented by meaningful symbolic constants, which are established by #define statements in the header file. For example, the value `CM_MAPPED_CONVERSATION` represents the integer 1. For the sake of portability and readability, use only the symbolic constants when writing programs.

Strings

All strings are in ASCII format when passed across the CPI-C interface.

Validity of Returned Parameters

The parameters returned by CPI-C are valid only if the CPI-C call is executed successfully, as indicated by a return code of `CM_OK`.

Information Provided for Windows Function Calls

WINDOWS

The following information is supplied for each of the Windows-specific function calls described in this chapter:

- The name of the call; unlike the CPI-C function calls, these calls do not have pseudonyms.
- A description of the call.
- The function prototype for the call, including the parameters used by the call and the data type for each parameter. The prototype of each function is declared in the file **wincpic.h**.

- A description of each supplied and returned parameter. The parameter names are pseudonyms. The actual variable names for these parameters are declared by the application program. The description includes the possible values of the parameter.
- Additional information describing the use of the call.

Accept_Conversation (cmaccp)

The Accept_Conversation call is issued by the invoked program to accept the incoming conversation and set certain conversation characteristics. For a list of initial conversation characteristics, see Chapter 2, "Writing CPI-C Applications," on page 19.

Upon successful execution of this call, CPI-C generates an 8-byte conversation identifier. This identifier is a required parameter for all other CPI-C calls issued by the invoked program during this conversation.

Function Call

```
void cmaccp (
             unsigned char CM_PTR      conversation_ID,
             CM_RETURN_CODE CM_PTR     return_code
            );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmaccp (
                           byte[]          conversation_ID,
                           CPICReturnCode  return_code
                          );
```

Supplied Parameters

There are no supplied parameters for this call.

Returned Parameters

After the call executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

conversation_ID
This is the identifier for the conversation. It is used by subsequent CPI-C calls.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_STATE_CHECK
This value indicates one of the following conditions:

- No incoming Allocate request was received within the timeout period specified in the configuration.
- The application has not specified any local TP names (or, for AIX or Linux systems,has released all specified names). The application must have at least one local TP name before issuing this call. For more information about specifying local TP names, see "Specifying the Local TP Name" on page 33.
- The application was started manually, but is defined in the invokable TP data file as nonqueued. A nonqueued TP is started automatically by CS/AIX in response to a conversation request (an incoming Attach); if you attempt to start it manually, the Accept_Conversation call will fail because there is no incoming Attach waiting for the application.

CM_PRODUCT_SPECIFIC_ERROR
See Appendix B, "Common Return Codes," on page 171.

State When Issued

The conversation must be in Reset state.

State Change

If the call is successful, the conversation changes to Receive state. If the call fails, the state remains unchanged.

Usage Notes

The TP name can be specified in a number of ways. For more information about specifying local TP names, see "Specifying the Local TP Name" on page 33. Before issuing Accept_Conversation, the program can issue Specify_Local_TP_Name to indicate one or more TP names for which it will accept incoming Allocates (these names are in addition to names defined in other ways such as the APPCTPN environment variable). If it specifies more than one TP name in this way, then it can use the Extract_TP_Name call (after Accept_Conversation returns) to determine which TP name the invoking program used.

AIX, LINUX

When Accept_Conversation returns CM_OK, a new conversation context is created for the conversation, and this becomes the program's current context.

Accept_Conversation always operates in blocking mode; that is, it always suspends until an incoming Allocate request is received. The following methods can be used to avoid unnecessary delays:

- Ensure that the invokable TP configuration for this application specifies a small timeout value, so that the Accept_Conversation call will return quickly (with *return_code* CM_PROGRAM_STATE_CHECK) if there is no incoming Allocate request, and then make the application retry Accept_Conversation later. The timeout value is specified in the invokable TP data file; refer to the *Communications Server for AIX Administration Guide* for more information.
- Instead of using Accept_Conversation, use Accept_Incoming, which can operate in nonblocking mode. Use the following sequence of calls:

– Initialize_For_Incoming (to obtain a conversation ID for the incoming conversation)
– Set_Processing_Mode (to set the *processing_mode* for this conversation ID to CM_NON_BLOCKING)
– Accept_Incoming

See the descriptions of these calls for more information.

Accept_Incoming (cmacci)

AIX, LINUX

The Accept_Incoming call is issued by the invoked program to accept an incoming conversation that has previously been initialized with Initialize_For_Incoming, and to set certain conversation characteristics. For a list of initial conversation characteristics, see "Initial Conversation Characteristics" on page 26.

Before issuing this call, the program can issue Set_Processing_Mode to set the processing mode for the conversation to CM_NON_BLOCKING. This ensures that the Accept_Incoming call and all subsequent CPI-C calls are issued in nonblocking mode.

Function Call

```
void cmacci (
             unsigned char CM_PTR      conversation_ID,
             CM_RETURN_CODE CM_PTR     return_code
            );
```

Function Call for Java CPI-C

```
public native void cmacci (
                           byte[]          conversation_ID,
                           CPICReturnCode  return_code
                          );
```

Supplied Parameters

The supplied parameter is:

conversation_ID
This is the identifier for the conversation that was returned on the previous Initialize_For_Incoming call. It is used to identify subsequent CPI-C calls on this conversation.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
The value specified by *conversation_ID* is not valid.

CM_PROGRAM_STATE_CHECK
One of the following occurred:

- The conversation specified by *conversation_ID* is not in Initialize-Incoming state.
- No incoming Allocate request was received within the timeout period specified in the configuration.
- The application has released the local TP name specified, for example, in the APPCTPN environment variable, and has not specified any additional local TP names. The application must have at least one local TP name before issuing this call. For more information about specifying local TP names, see "Specifying the Local TP Name" on page 33.
- The application was started manually, but is defined in the invokable TP data file as nonqueued. A nonqueued TP is started automatically by CS/AIX in response to an incoming Attach; if you attempt to start it manually, the Accept_Incoming call will fail because there is no incoming Attach waiting for the application.

For an explanation of the following return codes, see Appendix B, "Common Return Codes," on page 171.

CM_OPERATION_INCOMPLETE
CM_OPERATION_NOT_ACCEPTED
CM_PRODUCT_SPECIFIC_ERROR

State When Issued

The conversation must be in Initialize-Incoming state.

State Change

If the call is successful, the conversation changes to Receive state. If the call fails, the state remains unchanged.

Usage Notes

Issuing Initialize_For_Incoming followed by Accept_Incoming is equivalent to issuing Accept_Conversation. The difference between the two methods of accepting a conversation is that Accept_Conversation always operates in blocking mode, whereas Accept_Incoming can operate in nonblocking mode. To accept a conversation in nonblocking mode, the program issues the following sequence of calls:

Initialize_For_Incoming (to obtain a conversation ID for the incoming conversation)
Set_Processing_Mode (to set the *processing_mode* for this conversation ID to CM_NON_BLOCKING)
Accept_Incoming

The TP name specified by the APPCTPN environment variable is normally the name used to match incoming Allocates with this program. Before issuing Accept_Incoming, the program can issue Specify_Local_TP_Name to indicate one or more TP names for which it will accept incoming Allocates (these names replace the name in APPCTPN). If it specifies more than one TP name in this way, then it can use the Extract_TP_Name call (after Accept_Incoming returns) to determine which

TP name the invoking program used. For more information about specifying local TP names, see "Specifying the Local TP Name" on page 33.

When Accept_Incoming returns CM_OK, a new conversation context is created for the conversation, and this becomes the program's current context. When Accept_Incoming returns CM_OPERATION_INCOMPLETE and a subsequent Wait_For_Conversation returns the completion of Accept_Incoming as CM_OK, a new conversation context is created for the conversation, but the program's current context is not changed. To use the new context, the program must issue Extract_Conversation_Context for this *conversation_ID* to get the value of the conversation's context, and Set_Conversation_Context to set the program's current context to this value.

Allocate (cmallc)

The Allocate call is issued by the invoking program to allocate a conversation with the partner program, using the current conversation characteristics. CPI-C can also allocate a session between the local LU and partner LU if one does not already exist.

The type of conversation allocated is based on the conversation type characteristic—mapped or basic.

Once the conversation has been allocated by this call, the following conversation characteristics cannot be changed:

- Conversation type
- Mode name
- Partner LU name
- Partner program name
- Return control
- Synchronization level
- Conversation security
- User ID
- Password

Function Call

```
void cmallc (
             unsigned char CM_PTR      conversation_ID,
             CM_RETURN_CODE CM_PTR     return_code
            );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmallc (
                           byte[]          conversation_ID,
                           CPICReturnCode  return_code
                          );
```

Supplied Parameters

The supplied parameter is:

conversation_ID
This is the conversation identifier. The value of this parameter is returned by the Initialize_Conversation call.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PARAMETER_ERROR
One of the following has occurred:

- The mode name derived from the side information or set by Set_Mode_Name is not valid.
- The mode name is one of the names reserved for SNA internal use (such as SNASVCMG); an application cannot use it.

CM_PROGRAM_PARAMETER_CHECK
The value specified by *conversation_ID* is not valid.

CM_PROGRAM_STATE_CHECK
The conversation is not in Initialize state.

CM_UNSUCCESSFUL
The conversation's return-control characteristic is set to CM_IMMEDIATE, and the local LU does not have an available contention winner session.

For an explanation of the following return codes, see Appendix B, "Common Return Codes," on page 171.

```
CM_ALLOCATE_FAILURE_NO_RETRY
CM_ALLOCATE_FAILURE_RETRY
CM_OPERATION_INCOMPLETE
CM_OPERATION_NOT_ACCEPTED
CM_PRODUCT_SPECIFIC_ERROR
```

State When Issued

The conversation must be in Initialize state.

State Change

State changes, summarized in Table 12, are based on the value of the *return_code* parameter.

Table 12. State Changes for the Allocate Call

return_code	**New state**
CM_OK	Send

Table 12. State Changes for the Allocate Call (continued)

return_code	**New state**
CM_ALLOCATE_FAILURE_NO_RETRY	Reset
CM_ALLOCATE_FAILURE_RETRY	
All others	No change

Usage Notes

To send the allocation request immediately, the invoking program can issue the Flush or Confirm call immediately after the Allocate call. Otherwise, the allocate request accumulates with other data in the local LU's send buffer until the buffer is full.

Because the allocation request is buffered and not sent immediately, the Allocate call may return CM_OK, but the partner LU may subsequently reject the allocation request generated by the Allocate call. This error is returned to the invoking program on a subsequent call.

If the conversation's synchronization level is set to CM_CONFIRM, the invoking program can immediately determine whether the allocation was successful by issuing the Confirm call after the Allocate call.

AIX, LINUX

The program's current context at the time the Allocate call is issued becomes the context for the new conversation when Allocate returns CM_OK. If the program is using multiple contexts (as a result of accepting multiple conversations), it must set the current context to the appropriate value before issuing the Allocate call.

Cancel_Conversation (cmcanc)

The Cancel_Conversation call ends a specified conversation, canceling any incomplete operation (a previous call that returned with CM_OPERATION_INCOMPLETE) on this conversation, and ends the session that the conversation was using. It is equivalent to the Deallocate call with the *deallocate_type* parameter set to CM_DEALLOCATE_ABEND, with the following differences:

- Deallocate cannot be used while an operation is incomplete; Cancel_Conversation can be used, and will cancel the outstanding call.
- Deallocate writes the log data, if any, to the local error log; Cancel_Conversation does not.

The results of the outstanding call are undefined, and will not be returned to the application. For example, if Cancel_Conversation is used to cancel an outstanding Send_Data call, some or all of the data may have been sent; if it is used to cancel Send_Error, an error indication may or may not have been sent to the partner program.

In Java CPI-C, nonblocking calls are not supported and so there cannot be an incomplete call outstanding. Cancel_Conversation is equivalent to Deallocate except that it does not write log data to the local error log.

Function Call

```
void cmcanc (
             unsigned char CM_PTR      conversation_ID,
             CM_RETURN_CODE CM_PTR     return_code
            );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmcanc (
                           byte[]          conversation_ID,
                           CPICReturnCode  return_code
                          );
```

Supplied Parameters

The supplied parameter is:

conversation_ID
This is the identifier for the conversation. The value of this parameter is returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

return_code
Possible values are:

CM_OK The call executed successfully. The specified conversation has been deallocated, and any outstanding call on this conversation has been canceled.

CM_PROGRAM_PARAMETER_CHECK
The value specified by *conversation_ID* is not valid.

CM_PRODUCT_SPECIFIC_ERROR
See Appendix B, “Common Return Codes,” on page 171.

State When Issued

The conversation can be in any state except Reset.

State Change

If the return code is CM_OK, the conversation state changes to Reset.

Usage Notes

The partner program is notified of the end of the conversation with the return code CM_DEALLOCATED_ABEND.

Check_For_Completion (cmchck)

AIX, LINUX

This function is not available in Java CPI-C.

The Check_For_Completion call checks whether a previous call that returned with CM_OPERATION_INCOMPLETE has since completed. This call returns immediately whether or not the previous call has completed; the application can then continue with other processing if the previous call has not yet completed, or call Wait_For_Conversation to obtain the results of the previous call if it has completed.

If the application is involved in multiple conversations, this call acts across all conversations, and returns a "successful" return code if a previous call has completed on any of them.

This call is not part of the standard CPI-C specification, and may not be available in other implementations. The standard method for obtaining the results of an outstanding call is to issue Wait_For_Conversation, which operates in blocking mode and waits until a call has completed.

Function Call

```
void cmchck (
             unsigned char CM_PTR      conversation_ID,
             CM_RETURN_CODE CM_PTR     return_code
            );
```

Supplied Parameters

There are no supplied parameters for this call.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

conversation_ID

The identifier for the conversation on which a previous outstanding call has completed. For more information, see "Usage Notes" on page 58.

This value is relevant only if the *return_code* parameter is set to CM_OK.

return_code

Possible values are:

CM_OK The call executed successfully. A previously outstanding call on the conversation specified by *conversation_ID* has completed.

CM_PROGRAM_STATE_CHECK

There are no previously incomplete calls outstanding. Either the application has not issued any calls that returned

CM_OPERATION_INCOMPLETE, or it has already issued Wait_For_Conversation to obtain the results of all such calls.

CM_UNSUCCESSFUL
There is at least one previously incomplete call outstanding, but none has yet completed. The application should continue with other processing and retry Check_For_Completion later. (This return code is different from CM_PROGRAM_STATE_CHECK.)

State When Issued

The call is not associated with a specific conversation, so the conversation state is not relevant. However, the application must have at least one conversation with an incomplete operation outstanding.

State Change

There is no state change.

Usage Notes

If the return code from Check_For_Completion is CM_OK, the application should call Wait_For_Conversation to obtain the results of the outstanding call.

If more than one call has completed since the application last issued Check_For_Completion or Wait_For_Conversation, issuing Check_For_Completion more than once does not necessarily return information about additional calls; it simply indicates that at least one call has completed, and therefore a subsequent Wait_For_Conversation call will return immediately and not block. Each Wait_For_Conversation call returns one incomplete operation; if there are multiple incomplete operations (on different conversations), the application can issue a further Check_For_Completion after Wait_For_Conversation to check whether further calls have completed.

The Wait_For_Conversation call does not necessarily return the information for the same call that was reported by Check_For_Completion.

Confirm (cmcfm)

The Confirm call sends the contents of the local LU's send buffer and a confirmation request to the partner program and waits for confirmation.

In response to the Confirm call, the partner program normally issues the Confirmed call to confirm that it has received the data without error. (If the partner program encounters an error, it issues the Send_Error call or uses the Deallocate call to abnormally deallocate the conversation.)

The program can issue the Confirm call only if the conversation's synchronization level is CM_CONFIRM.

Function Call

```
void cmcfm (
            unsigned char CM_PTR                  conversation_ID,
            CM_Request_to_Send_Received CM_PTR    request_to_send_received,
            CM_RETURN_CODE CM_PTR                 return_code
           );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmcfm (
                          byte[]                          conversation_ID,
                          CPICControlInformationReceived  request_to_send_received,
                          CPICReturnCode                  return_code
                         );
```

Supplied Parameters

The supplied parameter is:

conversation_ID
This is the identifier for the conversation. The value of this parameter is returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

request_to_send_received
This is the request-to-send-received indicator. Possible values are:

CM_REQ_TO_SEND_RECEIVED
The partner program has issued the Request_To_Send call, which requests the local program to change the conversation to Receive state.

CM_REQ_TO_SEND_NOT_RECEIVED
The partner program has not issued the Request_To_Send call.

This value is not relevant if the *return_code* parameter is set to one of the following:

- CM_PROGRAM_PARAMETER_CHECK
- CM_PROGRAM_STATE_CHECK

return_code
Possible values are:

CM_OK The call executed successfully. The partner program has issued the Confirmed call.

CM_PROGRAM_PARAMETER_CHECK
One of the following has occurred:

- The value specified by *conversation_ID* is not valid.

- The local program attempted to use the Confirm call in a conversation with a synchronization level of CM_NONE. The synchronization level must be CM_CONFIRM.

CM_PROGRAM_STATE_CHECK
One of the following has occurred:

- The conversation was not in Send or Send-Pending state.
- The basic conversation for the local program was in Send state, and the local program did not finish sending a logical record.

For an explanation of the following return codes, see Appendix B, "Common Return Codes," on page 171.

CM_CONVERSATION_TYPE_MISMATCH
CM_DEALLOCATED_ABEND
CM_DEALLOCATED_ABEND_SVC
CM_DEALLOCATED_ABEND_TIMER
CM_OPERATION_INCOMPLETE
CM_OPERATION_NOT_ACCEPTED
CM_PIP_NOT_SPECIFIED_CORRECTLY
CM_PRODUCT_SPECIFIC_ERROR
CM_PROGRAM_ERROR_PURGING
CM_RESOURCE_FAILURE_NO_RETRY
CM_RESOURCE_FAILURE_RETRY
CM_SECURITY_NOT_VALID
CM_SVC_ERROR_PURGING
CM_SYNC_LVL_NOT_SUPPORTED_PGM
CM_SYNC_LVL_NOT_SUPPORTED_LU
CM_TP_NOT_AVAILABLE_NO_RETRY
CM_TP_NOT_AVAILABLE_RETRY
CM_TPN_NOT_RECOGNIZED

State When Issued

The conversation can be in Send or Send-Pending state.

State Change

State changes, summarized in Table 13, are based on the value of the *return_code* parameter.

Table 13. State Changes for the Confirm Call

return_code	**New state**
CM_OK (Call issued in Send state)	No change
CM_OK (Call issued in Send-Pending state)	Send
CM_PROGRAM_ERROR_PURGING CM_SVC_ERROR_PURGING	Receive

Table 13. State Changes for the Confirm Call (continued)

return_code	**New state**
CM_CONVERSATION_TYPE_MISMATCH	Reset
CM_PIP_NOT_SPECIFIED_CORRECTLY	
CM_SECURITY_NOT_VALID	
CM_SYNC_LEVEL_NOT_SUPPORTED_PGM	
CM_SYNC_LEVEL_NOT_SUPPORTED_LU	
CM_TPN_NOT_RECOGNIZED	
CM_TP_NOT_AVAILABLE_NO_RETRY	
CM_TP_NOT_AVAILABLE_RETRY	
CM_RESOURCE_FAILURE_NO_RETRY	
CM_RESOURCE_FAILURE_RETRY	
CM_DEALLOCATED_ABEND	
CM_DEALLOCATED_ABEND_SVC	
CM_DEALLOCATED_ABEND_TIMER	
All others	No change

Usage Notes

The Confirm call waits for a response from the partner program. A response is generated by one of the following CPI-C calls in the partner program:

- Confirmed
- Send_Error
- Deallocate with the conversation's deallocate type set to CM_DEALLOCATE_ABEND

Confirmed (cmcfmd)

The Confirmed call replies to a confirmation request from the partner program. It informs the partner program that the local program has not detected an error in the received data.

Because the program issuing the confirmation request waits for a confirmation, the Confirmed call synchronizes the processing of the two programs.

Function Call

```
void cmcfmd (
             unsigned char CM_PTR      conversation_ID,
             CM_RETURN_CODE CM_PTR     return_code
            );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmcfmd (
                             byte[]          conversation_ID,
                             CPICReturnCode  return_code
                           );
```

Supplied Parameters

The supplied parameter is:

conversation_ID
This is the identifier for the conversation. The value of this parameter is returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
The value specified by *conversation_ID* is not valid.

CM_PROGRAM_STATE_CHECK
When the program issued this call the conversation was not in Confirm, Confirm-Send, or Confirm-Deallocate state.

For an explanation of the following return codes, see Appendix B, "Common Return Codes," on page 171.

```
CM_OPERATION_INCOMPLETE
CM_OPERATION_NOT_ACCEPTED
CM_PRODUCT_SPECIFIC_ERROR
```

State When Issued

The conversation must be in one of the following states when the program issues this call:

- Confirm
- Confirm-Send
- Confirm-Deallocate

State Change

The new state is determined by the old state: the state of the conversation when the local program issued the Confirmed call. The old state is indicated by the value of the *status_received* parameter of the preceding Receive call. Table 14 summarizes the possible state changes when *return_code* is set to `CM_OK`.

Table 14. State Changes for the Confirmed Call

Old state	New state
Confirm	Receive
Confirm-Send	Send
Confirm-Deallocate	Reset

Other return codes result in no state change.

Usage Notes

The following sections describe additional usage information for the Confirmed call.

Sources of Confirmation Requests

A confirmation request is issued by one of the following calls in the partner program:

- Confirm
- Prepare_To_Receive if the prepare-to-receive type is set to either CM_PREP_TO_RECEIVE_CONFIRM or CM_PREP_TO_RECEIVE_SYNC_LEVEL and the conversation's synchronization level is set to CM_CONFIRM
- Deallocate if the deallocate type is set to CM_DEALLOCATE_CONFIRM or to CM_DEALLOCATE_SYNC_LEVEL and the conversation's synchronization level is set to CM_CONFIRM
- Send_Data under the following circumstances:
  - The send type is set to CM_SEND_AND_CONFIRM
  - The send type is set to CM_SEND_AND_PREP_TO_RECEIVE and the prepare-to-receive type is set to CM_PREP_TO_RECEIVE_CONFIRM
  - The send type is set to CM_SEND_AND_PREP_TO_RECEIVE, the prepare-to-receive type is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and the synchronization level is set to CM_CONFIRM
  - The send type is set to CM_SEND_AND_DEALLOCATE and the deallocate type is set to CM_DEALLOCATE_CONFIRM
  - The send type is set to CM_SEND_AND_DEALLOCATE, the deallocate type is set to CM_DEALLOCATE_SYNC_LEVEL and the synchronization level is set to CM_CONFIRM

Receiving Confirmation Requests

A confirmation request is received by the local program through the *status_received* parameter of the Receive call. The local program can issue the Confirmed call only if the *status_received* parameter is set to one of the following values:

- CM_CONFIRM_RECEIVED
- CM_CONFIRM_SEND_RECEIVED
- CM_CONFIRM_DEALLOC_RECEIVED

Convert_Incoming (cmcnvi)

The Convert_Incoming call converts a character string from EBCDIC to ASCII. If the partner application sends data consisting of EBCDIC character strings, the local application can use Convert_Incoming to convert these strings to ASCII. (CPI-C parameters other than the data in Send_Data and Receive calls, such as *mode_name* and *TP_name*, are always specified in ASCII and do not require conversion.)

Function Call

```
void cmcnvi (
             unsigned char CM_PTR      string,
             CM_INT32 CM_PTR           string_length,
             CM_RETURN_CODE CM_PTR     return_code
            );
```

Function Call for Java CPI-C

```
public native void cmcnvi (
                             byte[]          string,
                             CPICLength      string_length,
                             CPICReturnCode  return_code
                           );
```

Supplied Parameters

The supplied parameters are:

string This is the EBCDIC string to be converted to ASCII. The CPI-C specification states that the string can contain any of the following characters (character set 640):

Uppercase A-Z, lowercase a-z, 0–9, the period (.) and space characters, and the special characters < + (& *) ; - / , % _ > ? : ' = "

In addition, CS/AIX CPI-C also accepts the following characters (which may not be supported by other CPI-C implementations):

! # $ @ \ { } ~

` (backward quotation mark)

| (solid vertical bar)

¦ (broken vertical bar)

¬ (NOT character)

¢ (cent)

The contents of this string (up to the number of characters specified in *string_length*) will be replaced by the ASCII string resulting from the conversion.

string_length
This is the number of characters to be converted (1–32,767).

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

string This is the ASCII string resulting from the conversion. This is valid up to the number of characters specified in *string_length*.

return_code
Possible values are:

CM_OK The call executed successfully. The string parameter now contains the converted ASCII string.

CM_PROGRAM_PARAMETER_CHECK
The *buffer_length* parameter specified a value that was not valid.

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

This call is not associated with a conversation.

State Change

There is no state change.

Usage Note

When data is being received in buffer format in a basic conversation (as specified by the Set_Fill call), the data buffer can contain multiple logical records, each consisting of a two-byte or four-byte header (LLID) followed by data. The application must extract and convert each data string separately (not including the headers). It must not attempt to convert the whole buffer in one operation because this will make the header values not valid.

Convert_Outgoing (cmcnvo)

The Convert_Outgoing call converts a character string from ASCII to EBCDIC. If the partner application requires data consisting of EBCDIC character strings, the local application can use Convert_Outgoing to convert data from ASCII to EBCDIC before sending it. (CPI-C parameters other than the data in Send_Data and Receive calls, such as *mode_name* and *TP_name*, are always specified in ASCII and do not require conversion.)

Function Call

```
void cmcnvo (
             unsigned char CM_PTR      string,
             CM_INT32 CM_PTR           string_length,
             CM_RETURN_CODE CM_PTR     return_code
            );
```

Function Call for Java CPI-C

```
public native void cmcnvo (
                             byte[]          string,
                             CPICLength      string_length,
                             CPICReturnCode  return_code
                           );
```

Supplied Parameters

The supplied parameters are:

string This is the ASCII string to be converted to EBCDIC. The CPI-C specification states that the string can contain any of the following characters (character set 640):

Uppercase `A-Z`, lowercase `a-z`, `0–9`, the period (`.`) and space characters, and the special characters `< + ( & * ) ; - / , % _ > ? : ' = "`

In addition, CS/AIX CPI-C also accepts the following characters (which may not be supported by other CPI-C implementations):

`! # $ @ \ { } ~`

` (backward quotation mark)

| (solid vertical bar)

¦ (broken vertical bar)

¬ (NOT character)

¢ (cent)

The contents of this string (up to the number of characters specified in *string_length*) will be replaced by the EBCDIC string resulting from the conversion.

string_length
This is the number of characters to be converted (1–32,767).

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

string This is the EBCDIC string resulting from the conversion. This is valid up to the number of characters specified in *string_length*.

return_code
Possible values are:

CM_OK The call executed successfully. The string parameter now contains the converted EBCDIC string.

CM_PROGRAM_PARAMETER_CHECK
The *buffer_length* parameter specified a value that was not valid.

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

This call is not associated with a conversation.

State Change

There is no state change.

Usage Note

When data is being sent in buffer format in a basic conversation (as specified by the Set_Fill call), the data buffer can contain multiple logical records, each consisting of a two-byte or four-byte header (LLID) followed by data. The application must convert each data string separately (not including the headers). It must not attempt to convert the whole buffer in one operation because this will make the header values not valid.

Deallocate (cmdeal)

The Deallocate call deallocates a conversation between two programs.

Before deallocating the conversation, this call performs the equivalent of either the Flush call or the Confirmed call, depending on the current conversation synchronization level and deallocate type. The deallocate type is set by the Set_Deallocate_Type call.

The partner program receives the deallocation notification through one of the following parameters:

- *status_received* = CM_CONFIRM_DEALLOC_RECEIVED
- *return_code* = CM_DEALLOCATED_NORMAL
- *return_code* = CM_DEALLOCATED_ABEND

After this call has successfully executed, the conversation ID is no longer valid.

Function Call

```
void cmdeal (
             unsigned char CM_PTR      conversation_ID,
             CM_RETURN_CODE CM_PTR     return_code
            );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmdeal (
                             byte[]          conversation_ID,
                             CPICReturnCode  return_code
                           );
```

Supplied Parameters

The supplied parameter is:

conversation_ID
This is the identifier for the conversation. The value of this parameter is returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

return_code
Possible values are:

CM_OK The call executed successfully, the conversation is deallocated.

CM_PROGRAM_PARAMETER_CHECK
The value specified by *conversation_ID* is not valid.

CM_PROGRAM_STATE_CHECK
The following state errors can occur when the deallocate type indicates a normal deallocation (CM_DEALLOCATE_SYNC_LEVEL, CM_DEALLOCATE_FLUSH, CM_DEALLOCATE_CONFIRM):

- The conversation is not in Send or Send-Pending state
- The conversation is in Send state, but the program did not finish sending a logical record

For an explanation of the following return codes, see Appendix B, "Common Return Codes," on page 171.

```
CM_OPERATION_INCOMPLETE
CM_OPERATION_NOT_ACCEPTED
CM_PRODUCT_SPECIFIC_ERROR
```

The following return codes can be returned when the deallocate type is set to CM_DEALLOCATE_CONFIRM or when it is set to CM_DEALLOCATE_SYNC_LEVEL and the conversation's synchronization level is set to CM_CONFIRM. For an explanation of these return codes, see Appendix B, "Common Return Codes," on page 171.

CM_CONVERSATION_TYPE_MISMATCH
CM_DEALLOCATED_ABEND
CM_DEALLOCATED_ABEND_SVC
CM_DEALLOCATED_ABEND_TIMER
CM_PIP_NOT_SPECIFIED_CORRECTLY
CM_SECURITY_NOT_VALID
CM_SVC_ERROR_PURGING
CM_SYNC_LVL_NOT_SUPPORTED_PGM
CM_SYNC_LVL_NOT_SUPPORTED_LU
CM_TP_NOT_AVAILABLE_NO_RETRY
CM_TP_NOT_AVAILABLE_RETRY
CM_TPN_NOT_RECOGNIZED
CM_PROGRAM_ERROR_PURGING
CM_RESOURCE_FAILURE_NO_RETRY
CM_RESOURCE_FAILURE_RETRY

State When Issued

The conversation can be in one of the states shown in Table 15 when the program issues the Deallocate call. This depends on the value of the conversation's *deallocate_type* parameter, set by the Set_Deallocate_Type call.

Table 15. Conversation States When Issuing the Deallocate Call

Deallocate type	Allowed state
CM_DEALLOCATE_FLUSH CM_DEALLOCATE_CONFIRM CM_DEALLOCATE_SYNC_LEVEL	Send or Send-Pending
CM_DEALLOCATE_ABEND	Any except Reset

State Change

State changes, summarized in Table 16, are based on the value of the *return_code* parameter.

Table 16. State Changes for the Deallocate Call

return_code	New state
CM_OK	Reset
CM_PROGRAM_ERROR_PURGING CM_SVC_ERROR_PURGING	Receive
CM_CONVERSATION_TYPE_MISMATCH CM_PIP_NOT_SPECIFIED_CORRECTLY CM_SECURITY_NOT_VALID CM_SYNC_LEVEL_NOT_SUPPORTED_PGM CM_SYNC_LEVEL_NOT_SUPPORTED_LU CM_TPN_NOT_RECOGNIZED CM_TP_NOT_AVAILABLE_NO_RETRY CM_TP_NOT_AVAILABLE_RETRY	Reset
CM_RESOURCE_FAILURE_NO_RETRY CM_RESOURCE_FAILURE_RETRY	Reset
CM_DEALLOCATED_ABEND CM_DEALLOCATED_ABEND_SVC CM_DEALLOCATED_ABEND_TIMER	Reset
All others	No change

Usage Notes

If the conversation's deallocate type is set to CM_DEALLOCATE_ABEND and the log data length is greater than 0 (zero), the local LU writes the log data (specified by the Set_Log_Data call) to the local error log file and to the partner LU. For information about log data, see "Set_Log_Data (cmsld)" on page 132.

After the Deallocate call has been executed, the log data length is set to 0 (zero) and the log data is set to null.

Delete_CPIC_Side_Information (xcmdsi)

This function is not available in Java CPI-C.

The Delete_CPIC_Side_Information call deletes a side information entry that the application has previously specified using Set_CPIC_Side_Information, or specifies that an entry in the configuration file is no longer available for use by this application. This entry is identified through the symbolic destination name.

This call is provided for compatibility with X/Open CPI-C and with the Windows CPI-C specification; it is not included in IBM CPI-C 2.0.

Function Call

```
void xcmdsi (
             unsigned char CM_PTR      key,
             unsigned char CM_PTR      sym_dest_name,
             CM_RETURN_CODE CM_PTR     return_code
            );
```

Supplied Parameters

The supplied parameters are:

key This parameter is ignored.

sym_dest_name
This parameter specifies the symbolic destination name of the entry to be deleted. It is an 8-byte ASCII character string and can contain any displayable characters.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
The *sym_dest_name* parameter has specified a nonexistent side information entry.

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

The call is not associated with a conversation.

State Change

There is no state change.

Usage Notes

This call does not modify the side information held in the configuration file; the change applies only to this application. CS/AIX stores the modified information in memory associated with this operating system process; the change is discarded when the process ends. For more details, see "Side Information" on page 30.

Extract_Conversation_Context (cmectx)

AIX, LINUX

The Extract_Conversation_Context call returns the context for a specified conversation. This enables the program to set its current context to the required value (using Set_Conversation_Context) before starting a new conversation, to ensure that the new conversation uses the same context.

Function Call

```
void cmectx (
             unsigned char CM_PTR      conversation_ID,
             unsigned char CM_PTR      context_ID,
             CM_INT32 CM_PTR           context_ID_length,
             CM_RETURN_CODE CM_PTR     return_code
            );
```

Function Call for Java CPI-C

```
public native void cmectx (
                            byte[]          conversation_ID,
                            byte[]          context_ID,
                            CPICLength      context_ID_length,
                            CPICReturnCode  return_code
                           );
```

Supplied Parameters

The supplied parameter is:

conversation_ID
: This is the identifier for the conversation. The value of this parameter was returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

context_ID
: This parameter contains the context of the specified conversation. It is valid only if the *return_code* parameter is `CM_OK`.

context_ID_length
This parameter contains the length of *context_ID* (1–32 bytes). It is valid only if the *return_code* parameter is CM_OK.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
The value specified by *conversation_ID* is not valid.

CM_PROGRAM_STATE_CHECK
The conversation specified by *conversation_ID* is in Initialize or Initialize-Incoming state.

For an explanation of the following return code, see Appendix B, "Common Return Codes," on page 171.
CM_PRODUCT_SPECIFIC_ERROR

State When Issued

The conversation can be in any state except Reset, Initialize, or Initialize-Incoming.

State Change

There is no state change.

Usage Notes

This call does not set the program's current context to the extracted value. The program must call Set_Conversation_Context to do this.

An application uses Extract_Conversation_Context, followed by Set_Conversation_Context, in the following situations:

- When it is involved in multiple conversations, and wants to allocate a new conversation using the same context as an existing conversation.
- When a CPI-C call that assigns a new context completes in nonblocking mode. For example, if Accept_Incoming completes immediately with *return_code* CM_OK, the program's current context is set to the context of the new conversation; however, if Accept_Incoming returns CM_OPERATION_INCOMPLETE, a subsequent Wait_For_Conversation that returns the result of Accept_Incoming does not change the program's current context. The program must use Extract_Conversation_Context and Set_Conversation_Context to set the current context to the correct value.

Extract_Conversation_Security_Type (xcecst)

This function is not available in Java CPI-C.

The Extract_Conversation_Security_Type call returns the security type for a specified conversation.

This call is provided for compatibility with X/Open CPI-C and with the Windows CPI-C specification; it is not included in IBM CPI-C 2.0.

Function Call

```
void xcecst (
             unsigned char CM_PTR                    conversation_ID,
             XC_CONVERSATION_SECURITY_TYPE CM_PTR    conversation_security_type,
             CM_RETURN_CODE CM_PTR                   return_code
            );
```

Supplied Parameters

The supplied parameter is:

conversation_ID
This is the identifier for the conversation. The value of this parameter was returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

Returned Parameters

After the verb executes, CS/AIX returns the following parameters:

conversation_security_type
This specifies the information the partner LU requires in order to validate access to the invoked program. Possible values are:

AIX, LINUX

CM_SECURITY_NONE
The invoked program uses no conversation security.

CM_SECURITY_SAME
This value is used when the invoked program, which has been invoked with a valid user ID and password, invokes another program (as illustrated in Chapter 1, "Concepts," on page 1. If program A invokes program B with a valid user ID and password, and program B in turn invokes program C, then if program B specifies the value CM_SECURITY_SAME, CPI-C will send an already-verified indicator to the LU for program C. This indicator tells program C not to require the password (if program C is configured to accept an already-verified indicator).

CM_SECURITY_PROGRAM
The invoked program uses conversation security and thus requires a user ID and password.

CM_SECURITY_PROGRAM_STRONG
As for CM_SECURITY_PROGRAM, except that the local node must not send the password across the network in clear text format. This value can be used only if the remote system supports password substitution.

WINDOWS

XC_SECURITY_NONE
Equivalent to CM_SECURITY_NONE

XC_SECURITY_SAME
Equivalent to CM_SECURITY_SAME

XC_SECURITY_PROGRAM
Equivalent to CM_SECURITY_PROGRAM

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
The value specified by *conversation_ID* is not valid.

State When Issued

The conversation can be in any state except Reset.

State Change

There is no state change.

Extract_Conversation_Security_User_ID (cmecsu)

WINDOWS

This call is the Windows CPI-C equivalent of the AIX or Linux CPI-C call Extract_Security_User_ID (cmesui). The two calls are used in exactly the same way, except that the names are different. For more information about Extract_Conversation_Security_User_ID, see "Extract_Security_User_ID (cmesui or cmecsu)" on page 83, and replace the AIX or Linux function name and pseudonym with the Windows function name and pseudonym as indicated.

Extract_Conversation_Security_User_ID (xcecsu)

This function is not available in Java CPI-C.

This call returns the user ID being used in a specified conversation.

The call provides compatibility for applications using the X/Open CPI-C definition. It has been incorporated into IBM CPI-C 2.0 as the call Extract_Security_User_ID (cmesui). Use cmesui whenever possible to enable greater portability of your program to other platforms.

The parameters on this call are identical to those on the cmesui call. For more information about cmesui, see "Extract_Security_User_ID (cmesui or cmecsu)" on page 83.

Extract_Conversation_State (cmecs)

The Extract_Conversation_State call returns the state of the specified conversation.

Function Call

```
void cmecs (
            unsigned char CM_PTR             conversation_ID,
            CM_CONVERSATION_STATE CM_PTR     conversation_state,
            CM_RETURN_CODE CM_PTR            return_code
           );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmecs (
                            byte[]                 conversation_ID,
                            CPICConversationState  conversation_state,
                            CPICReturnCode         return_code
                          );
```

Supplied Parameters

The supplied parameter is:

conversation_ID
: This is the identifier for the conversation. The value of this parameter was returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

conversation_state
: This specifies the conversation state. Possible values are:

  CM_INITIALIZE_STATE
  CM_INITIALIZE_INCOMING_STATE
  CM_SEND_STATE
  CM_RECEIVE_STATE
  CM_SEND_PENDING_STATE
  CM_CONFIRM_STATE
  CM_CONFIRM_SEND_STATE
  CM_CONFIRM_DEALLOCATE_STATE

return_code
: Possible values are:

  CM_OK The call executed successfully.

  CM_PROGRAM_PARAMETER_CHECK
  The value specified by *conversation_ID* is not valid.

  For an explanation of the following return codes, see Appendix B, "Common Return Codes," on page 171.

  CM_OPERATION_INCOMPLETE
  CM_PRODUCT_SPECIFIC_ERROR

State When Issued

The conversation can be in any state except Reset.

State Change

There is no state change.

Extract_Conversation_Type (cmect)

The Extract_Conversation_Type call returns the conversation type (mapped or basic) of the specified conversation.

Function Call

```
void cmect (
            unsigned char CM_PTR             conversation_ID,
            CM_CONVERSATION_TYPE CM_PTR      conversation_type,
            CM_RETURN_CODE CM_PTR            return_code
           );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmect (
                           byte[]               conversation_ID,
                           CPICConversationType conversation_type,
                           CPICReturnCode       return_code
                         );
```

Supplied Parameters

The supplied parameter is:

conversation_ID
This is the identifier for the conversation. The value of this parameter is returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

conversation_type
This parameter specifies the conversation type. Possible values are:
`CM_BASIC_CONVERSATION`
`CM_MAPPED_CONVERSATION`

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
The value specified by *conversation_ID* is not valid.

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

The conversation can be in any state except Reset.

State Change

There is no state change.

Extract_CPIC_Side_Information (xcmesi)

This function is not available in Java CPI-C.

The Extract_CPIC_Side_Information call returns the side information for an entry number or symbolic destination name.

This call is provided for compatibility with X/Open CPI-C and with the Windows CPI-C specification; it is not included in IBM CPI-C 2.0.

Function Call

```
  void xcmesi  (
                CM_INT32 CM_PTR             entry_number,
                unsigned char CM_PTR        sym_dest_name,
                SIDE_INFO CM_PTR            side_info_entry,
                CM_INT32 CM_PTR             side_info_entry_length,
                CM_RETURN_CODE CM_PTR       return_code
               );
typedef struct side_info_entry
{
 unsigned char    sym_dest_name[8];           /* symbolic destination name   */
 unsigned char    partner_LU_name[17];        /* Fully qualified partner LU  */
                                              /*  name                       */
 unsigned char    reserved[3];                /* Reserved                    */
 XC_TP_NAME_TYPE  TP_name_type;               /* TP name type                */
 unsigned char    TP_name[64];                /* TP name                     */
 unsigned char    mode_name[8];               /* Mode name                   */
 XC_CONVERSATION_SECURITY_TYPE
                  conversation_security_type; /* Conversation security type  */
 unsigned char    security_user_ID[8];        /* User ID                     */
 unsigned char    security_password[8];       /* Password                    */
} SIDE_INFO;
```

Supplied Parameters

The supplied parameters are:

entry_number
This parameter is ignored.

sym_dest_name
This parameter specifies the symbolic destination name to search for. It is an 8-byte ASCII character string and can contain any displayable characters.

side_info_entry_length

AIX, LINUX

This value must always be set to sizeof(SIDE_INFO).

WINDOWS

This value must always be set to 124.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

side_info_entry
This parameter specifies the contents of a side information entry, as follows.

side_info_entry.sym_dest_name
Symbolic destination name which identifies the side information entry. The parameter *sym_dest_name* is an 8-byte ASCII character string and can contain any displayable characters.

side_info_entry.partner_LU_name
Fully qualified name of the partner LU. This name is composed of two character strings each of 1–8 bytes, concatenated by a dot.

side_info_entry.TP_name_type
The type of the target TP (the valid characters for a TP name are determined by the TP type). Possible values are:

XC_APPLICATION_TP
Application TP. All characters in the TP name must be valid ASCII characters.

XC_SNA_SERVICE_TP
Service TP. The TP name must be specified as an 8–character ASCII string representing the hexadecimal digits of a 4-character name. For example, if the hexadecimal representation of the name is 0x21F0F0F8, set the *TP_name* parameter to the 8–character string “21F0F0F8”.

The first character (represented by two bytes) must be a hexadecimal value in the range 0x0–0x3F, excluding 0x0E and 0x0F; the remaining characters (each represented by two bytes) must be valid EBCDIC characters.

side_info_entry.TP_name
TP name of the target TP.

side_info_entry.mode_name
Name of the mode used to access the target TP.

side_info_entry.conversation_security_type
Specifies whether the target TP uses conversation security. Possible values are:

XC_SECURITY_NONE
The target TP does not use conversation security.

XC_SECURITY_PROGRAM
The target TP uses conversation security. The *security_user_ID* and *security_password* parameters specified below will be used to access the target TP.

XC_SECURITY_SAME
The target TP uses conversation security, and can accept an "already verified" indicator from the local TP. (This indicates that the local TP was itself invoked by another TP, and has verified the security user ID and password supplied by this TP.) The *security_user_ID* parameter specified below will be used to access the target TP; no password is required.

side_info_entry.security_user_ID
User ID used to access the partner TP. This parameter is not required if the *conversation_security_type* parameter is set to `XC_SECURITY_NONE`.

For compatibility with X/Open CPI-C, this verb only returns eight characters for the user ID, although security user IDs can be up to 10 characters. To ensure that you obtain the complete user ID, you should extract it explicitly using the Extract_Security_User_ID call (Extract_Conversation_Security_User_ID for Windows systems), instead of relying on the value returned here.

side_info_entry.security_password
This parameter is reserved; password information is never returned to the application.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
One of the following has occurred:

- The *sym_dest_name* parameter is not valid
- The *side_info_entry_length* parameter is not set to `sizeof(SIDE_INFO)`

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

This call is not associated with a conversation.

State Change

There is no state change.

Usage Notes

If the security user ID in the side information is not set, the security user ID field is returned filled with spaces.

Extract_Local_LU_Name (cmelln)

The Extract_Local_LU_Name call returns the alias of the local LU for a specified conversation.

This call is not part of the standard CPI-C specification, and may not be available in other implementations. In particular, it is not supported in other Java CPI-C implementations.

Function Call

```
void cmelln   (
               unsigned char CM_PTR         conversation_ID,
               unsigned char CM_PTR         lu_alias,
               CM_RETURN_CODE CM_PTR        return_code
              );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmelln (
                             byte[]          conversation_ID,
                             byte[]          lu_alias,
                             CPICReturnCode  return_code
                           );
```

Supplied Parameters

The supplied parameter is:

conversation_ID
This is the identifier for the conversation. The value of this parameter was returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

lu_alias
LU alias of the local LU.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
The value specified by *conversation_ID* is not valid.

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, “Common Return Codes,” on page 171.

State When Issued

The conversation can be in any state except Reset.

State Change

There is no state change.

Usage Notes

The LU alias returned by this call does not have to be set by Set_Local_LU_Name, as described in "Set_Local_LU_Name (cmslln)" on page 131. Any of the methods described in "Specifying the Local LU" on page 34 can be used.

Extract_Maximum_Buffer_Size (cmembs)

AIX, LINUX

The Extract_Maximum_Buffer_Size call returns the maximum size of a CPI-C data buffer. This defines the maximum amount of data that can be sent in one Send_Data call or received in one Receive call.

CS/AIX CPI-C always uses a data buffer size of 32,767 bytes. However, for compatibility with other CPI-C implementations (or with future versions of CS/AIX), an application should not rely on this value, and should use this call to determine the largest buffer size it can use.

Function Call

```
void cmembs (
             CM_INT32 CM_PTR           maximum_buffer_size,
             CM_RETURN_CODE CM_PTR     return_code
            );
```

Function Call for Java CPI-C

```
public native void cmembs (
                             CPICLength      maximum_bufer_size,
                             CPICReturnCode  return_code
                           );
```

Supplied Parameters

There are no supplied parameters for this call.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

maximum_buffer_size
This parameter specifies the length of the data buffer.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

This call is not associated with any conversation.

State Change

There is no state change.

Extract_Mode_Name (cmemn)

The Extract_Mode_Name call returns the mode name and mode name length for a specified conversation.

Function Call

```
void 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
           );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmemn (
                           byte[]          conversation_ID,
                           byte[]          mode_name,
                           CPICLength      mode_name_length,
                           CPICReturnCode  return_code
                         );
```

Supplied Parameters

The supplied parameter is:

conversation_ID
This is the identifier for the conversation. The value of this parameter was returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

mode_name
This parameter specifies the starting address of the mode name.

mode_name_length
This parameter specifies the length of the mode name.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
The value specified by *conversation_ID* is not valid.

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

The conversation can be in any state except Reset.

State Change

There is no state change.

Extract_Partner_LU_Name (cmepln)

The Extract_Partner_LU_Name call returns the partner LU name and partner LU name length for a specified conversation. This can be an alias name of up to eight bytes or a fully qualified network name of up to 17 bytes.

Function Call

```
void 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
            );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmepln (
                             byte[]          conversation_ID,
                             byte[]          partner_LU_name,
                             CPICLength      partner_LU_name_length,
                             CPICReturnCode  return_code
                           );
```

Supplied Parameters

The supplied parameter is:

conversation_ID
This is the identifier for the conversation. The value of this parameter is returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

partner_LU_name
This parameter specifies the variable containing the partner LU name. (The program must supply a pointer to a suitable variable.)

partner_LU_name_length
This parameter specifies the length of the partner LU name.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
The value specified by *conversation_ID* is not valid.

CM_PROGRAM_STATE_CHECK
The conversation specified by *conversation_ID* is in Initialize-Incoming state.

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

The conversation can be in any state except Reset or Initialize-Incoming.

State Change

There is no state change.

Extract_Security_User_ID (cmesui or cmecsu)

The Extract_Security_User_ID call returns the user ID being used in a specified conversation.

WINDOWS

This call is named Extract_Conversation_Security_User_ID, with the pseudonym cmecsu, for compatibility with the Windows CPI-C interface.

Function Call

```
void cmesui (
             unsigned char CM_PTR      conversation_ID,
             unsigned char CM_PTR      security_user_ID,
             CM_INT32 CM_PTR           security_user_ID_length,
             CM_RETURN_CODE CM_PTR     return_code
            );
```

WINDOWS

For Windows systems, replace cmesui with cmecsu.

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmesui (
                             byte[]          conversation_ID,
                             byte[]          security_user_ID,
                             CPICLength      security_user_ID_length,
                             CPICReturnCode  return_code
                           );
```

Supplied Parameters

The supplied parameter is:

conversation_ID
This is the identifier for the conversation. The value of this parameter is returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

Returned Parameters

After the verb executes, CS/AIX returns the following parameters:

security_user_ID
This specifies the user ID used to establish the conversation.

security_user_ID_length
This specifies the length of *security_user_ID*.

The range for this value is 1–10 characters (AIX or Linux systems), or 1–8 characters (Windows systems). If the *security_user_ID_length* is set to `0` (zero), the *security_user_ID_length* parameter is ignored; this is equivalent to setting *security_user_ID* to a null string.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
The value specified by *conversation_ID* is not valid.

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

The conversation can be in any state except Reset.

State Change

There is no state change.

Usage Notes

The *security_user_ID* value is not blank-padded. It is meaningful only up to *security_user_ID_length*.

Extract_Sync_Level (cmesl)

The Extract_Sync_Level call returns the synchronization level for a specified conversation.

Function Call

```
void cmesl (
           unsigned char CM_PTR       conversation_ID,
           CM_INT32 CM_PTR            sync_level,
           CM_RETURN_CODE CM_PTR      return_code
          );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmesl (
                            byte[]          conversation_ID,
                            CPICSyncLevel   sync_level,
                            CPICReturnCode  return_code
                          );
```

Supplied Parameters

The supplied parameter is:

conversation_ID
This is the identifier for the conversation. The value of this parameter was returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

sync_level
This parameter indicates the synchronization level of the conversation. Possible values are:

CM_NONE
The programs will not perform confirmation processing.

CM_CONFIRM
The programs can perform confirmation processing.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
The value specified by *conversation_ID* is not valid.

CM_PROGRAM_STATE_CHECK
The conversation specified by *conversation_ID* is in Initialize-Incoming state.

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

The conversation can be in any state except Reset or Initialize-Incoming.

State Change

There is no state change.

Extract_TP_Name (cmetpn)

The Extract_TP_Name call returns the TP name and TP name length of the invoked TP for a specified conversation.

An application that has used the Specify_Local_TP_Name call to accept incoming Allocates for more than one TP name, and has subsequently issued Accept_Conversation or Accept_Incoming to accept an incoming Allocate, can use this call to determine which TP name was specified on the incoming Allocate.

Function Call

```
void 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
            );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmetpn (
                           byte[]          conversation_ID,
                           byte[]          TP_name,
                           CPICLength      TP_name_length,
                           CPICReturnCode  return_code
                          );
```

Supplied Parameters

The supplied parameter is:

conversation_ID
This is the identifier for the conversation. The value of this parameter was returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

TP_name
This parameter specifies the starting address of the TP name.

TP_name_length
This parameter specifies the length of the TP name.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
The value specified by *conversation_ID* is not valid.

CM_PROGRAM_STATE_CHECK
The conversation is in Reset or Initialize-Incoming state.

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

The conversation can be in any state except Reset or Initialize-Incoming.

State Change

There is no state change.

Flush (cmflus)

The Flush call sends the contents of the local LU's send buffer to the partner LU (and program). If the send buffer is empty, no action takes place.

Sources of Buffered Data

Data processed by the Send_Data call accumulates in the local LU's send buffer until one of the following happens:

- The local program issues the Flush call or other call that flushes the LU's send buffer. (Some send types, set by the Set_Send_Type call, include flush functionality.)
- The buffer is full.

The allocation request generated by the Allocate call and error information generated by the Send_Error call are also buffered.

Function Call

```
void cmflus (
             unsigned char CM_PTR      conversation_ID,
             CM_RETURN_CODE CM_PTR     return_code
            );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmflus (
                             byte[]          conversation_ID,
                             CPICReturnCode  return_code
                           );
```

Supplied Parameters

The supplied parameter is:

conversation_ID
This is the identifier for the conversation.

The value of this parameter is returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
The value specified by *conversation_ID* is not valid.

CM_PROGRAM_STATE_CHECK
The conversation was not in Send or Send-Pending state when the program issued this call.

For an explanation of the following return codes, see Appendix B, "Common Return Codes," on page 171.

```
CM_OPERATION_INCOMPLETE
CM_OPERATION_NOT_ACCEPTED
CM_PRODUCT_SPECIFIC_ERROR
```

State When Issued

The conversation must be in Send or Send-Pending state.

State Change

If the call completes successfully (*return_code* = CM_OK), the conversation is in Send state.

Other return codes result in no state change.

Initialize_Conversation (cminit)

The Initialize_Conversation call is issued by the invoking program to obtain an 8-byte conversation ID and to set the initial values for the conversation's characteristics.

The initial values are CPI-C defaults or are derived from side information associated with the symbolic destination name. For more information about initial values and side information, see Chapter 2, "Writing CPI-C Applications," on page 19.

Upon successful execution of this call, CPI-C generates a conversation identifier. This identifier is a required parameter for all other CPI-C calls issued for this conversation by the invoking program.

Initial values can be changed by the Set_* calls.

Function Call

```
void cminit (
             unsigned char CM_PTR      conversation_ID,
             unsigned char CM_PTR      sym_dest_name,
             CM_RETURN_CODE CM_PTR     return_code
            );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cminit (
                            byte[]          conversation_ID,
                            String          sym_dest_name,
                            CPICReturnCode  return_code
                           );
```

Supplied Parameters

The supplied parameter is:

sym_dest_name

This parameter specifies the symbolic destination name—the name associated with a side information entry loaded from the CS/AIX configuration file or defined by Set_CPIC_Side_Information calls.

The parameter is an 8-byte ASCII character string and can contain any displayable characters. This parameter can also be set to eight spaces. In this case, the invoking program must issue the following calls before issuing the Allocate call:

- Set_Mode_Name
- Set_Partner_LU_Name
- Set_TP_Name

For more details about the side information entry, see "Set_CPIC_Side_Information (xcmssi)" on page 122.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

conversation_ID
This is the identifier for the conversation. It is used by subsequent CPI-C calls.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
One of the following occurred:

- The value specified by *sym_dest_name* does not match a symbolic destination name defined in the configuration file or one specified by the program using Set_CPIC_Side_Information.
- The *conversation_ID* parameter is not valid.

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

The conversation is in Reset state.

State Change

If the *return_code* is CM_OK, the conversation changes to Initialize state. For other return codes, the conversation state remains unchanged.

Usage Notes

If the side information contains a value that is not valid for a conversation characteristic, or if a Set_* call sets it to a value that is not valid, then the error is returned on the Allocate call.

Initialize_For_Incoming (cminic)

AIX, LINUX

The Initialize_For_Incoming call is issued by the invoked program to obtain an 8-byte conversation ID. The program then accepts the conversation using the Accept_Incoming call.

Issuing Initialize_For_Incoming followed by Accept_Incoming is equivalent to issuing Accept_Conversation. The difference is that Set_Processing_Mode can be issued between Initialize_For_Incoming and Accept_Incoming to ensure that Accept_Incoming operates in nonblocking mode, whereas Accept_Conversation always operates in blocking mode.

Function Call

```
void cminic (
             unsigned char CM_PTR      conversation_ID,
             CM_RETURN_CODE CM_PTR     return_code
            );
```

Function Call for Java CPI-C

```
public native void cminic (
                            byte[]          conversation_ID,
                            CPICReturnCode  return_code
                           );
```

Supplied Parameters

There are no supplied parameters for this call.

Returned Parameters

After the verb executes, CS/AIX returns the following parameters:

conversation_ID
This is the identifier for the conversation. It is used by subsequent CPI-C calls.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

The conversation is in Reset state.

State Change

If the *return_code* is CM_OK, the conversation changes to Initialize state. Otherwise the conversation state remains unchanged.

Prepare_To_Receive (cmptr)

The Prepare_To_Receive call changes the state of the conversation for the local program from Send to Receive. Before changing the conversation state, this call performs the equivalent of one of the following:

- The Flush call, sending the contents of the local LU's send buffer to the partner LU (and program), if either of the following conditions is true:
  - The conversation's prepare-to-receive type is set to CM_PREP_TO_RECEIVE_FLUSH
  - The prepare-to-receive type is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and the conversation's synchronization level is set to CM_NONE
- The Confirm call, sending the contents of the local LU's send buffer and a confirmation request to the partner program, if either of the following conditions is true:

- The conversation's 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 the conversation's synchronization level is set to CM_CONFIRM

The prepare-to-receive type is set by the Set_Prepare_To_Receive_Type call; the synchronization level is set by the Set_Sync_Level call.

After this call has successfully executed, the local program can receive data.

Function Call

```
void cmptr (
            unsigned char CM_PTR       conversation_ID,
            CM_RETURN_CODE CM_PTR      return_code
           );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmptr (
                           byte[]          conversation_ID,
                           CPICReturnCode  return_code
                         );
```

Supplied Parameters

The supplied parameter is:

conversation_ID
This is the identifier for the conversation. The value of this parameter is returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
The value specified by *conversation_ID* is not valid.

CM_PROGRAM_STATE_CHECK
One of the following has occurred:

- The conversation state is not Send or Send-Pending
- The basic conversation is in Send state. However, the program did not finish sending a logical record

For an explanation of the following return codes, see Appendix B, "Common Return Codes," on page 171.

CM_OPERATION_INCOMPLETE
CM_OPERATION_NOT_ACCEPTED
CM_PRODUCT_SPECIFIC_ERROR

The following return codes can occur if the conversation's prepare-to-receive type is set to CM_PREP_TO_RECEIVE_CONFIRM or if the prepare-to-receive type is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, and the conversation's synchronization level is set to CM_CONFIRM. For an explanation of them, see Appendix B, "Common Return Codes," on page 171.

CM_CONVERSATION_TYPE_MISMATCH
CM_DEALLOCATED_ABEND
CM_DEALLOCATED_ABEND_SVC
CM_DEALLOCATED_ABEND_TIMER
CM_PIP_NOT_SPECIFIED_CORRECTLY
CM_PROGRAM_ERROR_PURGING
CM_RESOURCE_FAILURE_NO_RETRY
CM_RESOURCE_FAILURE_RETRY
CM_SECURITY_NOT_VALID
CM_SVC_ERROR_PURGING
CM_SYNC_LVL_NOT_SUPPORTED_PGM
CM_SYNC_LVL_NOT_SUPPORTED_LU
CM_TPN_NOT_RECOGNIZED
CM_TP_NOT_AVAILABLE_NO_RETRY
CM_TP_NOT_AVAILABLE_RETRY

State When Issued

The conversation can be in Send or Send-Pending state.

State Change

State changes, summarized in Table 17, are based on the value of the *return_code* parameter.

Table 17. State Changes for the Prepare_To_Receive Call

return_code	**New state**
CM_OK	Receive
CM_PROGRAM_ERROR_PURGING	Receive
CM_SVC_ERROR_PURGING	
CM_CONVERSATION_TYPE_MISMATCH	Reset
CM_PIP_NOT_SPECIFIED_CORRECTLY	
CM_SECURITY_NOT_VALID	
CM_SYNC_LEVEL_NOT_SUPPORTED_PGM	
CM_SYNC_LEVEL_NOT_SUPPORTED_LU	
CM_TPN_NOT_RECOGNIZED	
CM_TP_NOT_AVAILABLE_NO_RETRY	
CM_TP_NOT_AVAILABLE_RETRY	
CM_DEALLOCATED_ABEND	Reset
CM_RESOURCE_FAILURE_NO_RETRY	
CM_RESOURCE_FAILURE_RETRY	
CM_DEALLOCATED_ABEND_SVC	Reset
CM_DEALLOCATED_ABEND_TIMER	
All others	No change

Usage Notes

The conversation does not change to Send (or Send-Pending) for the partner program until the partner program receives one of the following values through the *status_received* parameter of the Receive call:

- CM_SEND_RECEIVED
- CM_CONFIRM_SEND_RECEIVED and replies with the Confirmed or Send_Error call

Receive (cmrcv)

The Receive call receives any data that is currently available from the partner program.

If no data is currently available and the receive type (set by the Set_Receive_Type call) is set to CM_RECEIVE_AND_WAIT, the local program waits for data to arrive. If the receive type is set to CM_RECEIVE_IMMEDIATE, the local program does not wait.

WINDOWS

If the Receive call is issued in nonblocking mode (specified by a previous Set_Processing_Mode call), the application can issue the following calls while Receive is outstanding:

- Request_To_Send
- Send_Error
- Test_Request_to_Send_Received
- Cancel_Conversation
- Deallocate

If the application uses one of these calls in nonblocking mode while the Receive call is outstanding, it must use Specify_Windows_Handle to enable CPI-C to return the results of nonblocking calls. It must not issue Wait_For_Conversation if another call is outstanding in addition to Receive; the results of this call are undefined if more than one call is outstanding on the same conversation.

How a Program Receives Data

The process for receiving data is as follows:

- The local program issues a Receive call until it finishes receiving a complete unit of data. The local program may need to issue the Receive call several times in order to receive a complete unit of data. The *data_received* parameter indicates whether the receipt of data is finished.

  The data received can be any of the following:

  – One data record transmitted in a mapped conversation
  – One logical record transmitted in a basic conversation with the conversation's fill characteristic set to CM_FILL_LL
  – A buffer of data received independent of its logical record format in a basic conversation with the fill characteristic set to CM_FILL_BUFFER

  Once a complete unit of data has been received, the local program can manipulate it.

- The local program determines the next action to be taken based on the control information received through the *status_received* parameter. The local program may need to issue the Receive call again to receive the control information.

The conversation type is set by the Set_Conversation_Type call; the fill characteristic is set by the Set_Fill call.

Function Call

```
void 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_INT32 CM_PTR                request_to_send_received,
            CM_RETURN_CODE CM_PTR          return_code
           );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmrcv (
                          byte[]                        conversation_ID,
                          byte[]                        buffer,
                          CPICLength                    requested_length,
                          CPICDataReceivedType          data_received,
                          CPICLength                    received_length,
                          CPICStatusReceived            status_received,
                          CPICControlInformationReceived request_to_send_received,
                          CPICReturnCode                return_code
                         );
```

Supplied Parameters

The supplied parameters are:

conversation_ID
This is the identifier for the conversation. The value of this parameter was returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

requested_length
This indicates the maximum number of bytes of data the local program is to receive.

The range for this value is `0–32,767`.

buffer This is the address of the buffer to contain the data received by the local program.

Returned Parameters

After the verb executes, CS/AIX returns the following parameters:

buffer The application's data buffer contains data if the following conditions are true:

- The *data_received* parameter is set to a value other than CM_NO_DATA_RECEIVED
- The *return_code* parameter is set to CM_OK or to CM_DEALLOCATED_NORMAL

data_received
This parameter indicates whether the program received data. The following are possible values. These codes are not relevant unless *return_code* is set to CM_OK or CM_DEALLOCATED_NORMAL.

CM_DATA_RECEIVED can be returned if the conversation's fill characteristic is set to CM_FILL_BUFFER, indicating that the program is receiving data independent of its logical format. The local program received data until *requested_length* or end of data was reached.

The end of the data is indicated by either of the following:

- A change to another conversation state, based on the *return_code*, *status_received*, and *data_received* parameters
- An error condition

If the conversation's receive type is set to CM_RECEIVE_IMMEDIATE, the data received can be less than *requested_length* if a smaller amount of data has arrived from the partner program.

CM_COMPLETE_DATA_RECEIVED
In a mapped conversation, this parameter indicates that the local program has received a complete data record or the last part of a data record.

In a basic conversation with the fill characteristic set to CM_FILL_LL, this value indicates that the local program has received a complete logical record or the end of a logical record.

CM_INCOMPLETE_DATA_RECEIVED
In a mapped conversation, this value indicates that the local program has received an incomplete data record, the *requested_length* parameter specified a value less than the length of the data record (or less than the remainder of the data record if this is not the first Receive call to read the record). The amount of data received is equal to the *requested_length* parameter.

In a basic conversation with the fill characteristic set to CM_FILL_LL, this value indicates that the local program has received an incomplete logical record. The amount of data received is equal to the *requested_length* parameter. (If the received data was truncated, the length of the data will be less than *requested_length*.)

Upon receiving this value, the local program normally reissues the Receive call to receive the next part of the record.

CM_NO_DATA_RECEIVED
The program did not receive data.

Note: If the *return_code* parameter is set to CM_OK, status information may be available through the *status_received* parameter.

received_length
This indicates the number of bytes of data the local program received on this Receive call. If the *return_code* or *data_received* parameter indicates that the program received no data, this value is not relevant.

status_received

This parameter indicates changes in the status of the conversation. These codes are not relevant unless *return_code* is set to CM_OK. Possible values are:

CM_NO_STATUS_RECEIVED
No conversation status change was received on this call.

CM_SEND_RECEIVED
For the partner program, the conversation has entered Receive state. For the local program, the conversation is now in Send state if no data was received on this call, or Send-Pending state if data was received on this call.

Upon receiving this value, the local program normally uses the Send_Data call to begin sending data.

CM_CONFIRM_DEALLOC_RECEIVED
The partner program has issued the Deallocate call with confirmation requested. For the local program the conversation is now in Confirm-Deallocate state.

Upon receiving this value, the local program normally issues the Confirmed call.

CM_CONFIRM_RECEIVED
The partner program has issued the Confirm call. For the local program the conversation is in Confirm state.

Upon receiving this value, the local program normally issues the Confirmed call.

CM_CONFIRM_SEND_RECEIVED
For the partner program, the conversation has entered Receive state and a request for confirmation has been received by the local program. For the local program, the conversation is now in Confirm-Send state.

The program normally responds by issuing the Confirmed call. Upon successful execution of the Confirmed call, the conversation changes to Send state for the local program.

request_to_send_received

This is the request-to-send-received indicator. Possible values are:

CM_REQ_TO_SEND_RECEIVED
The partner program has issued the Request_To_Send call, which requests the local program to change the conversation to Receive state.

CM_REQ_TO_SEND_NOT_RECEIVED
The partner program has not issued the Request_To_Send call.

This value is not relevant if the *return_code* parameter is set to one of the following:

- CM_PROGRAM_PARAMETER_CHECK
- CM_PROGRAM_STATE_CHECK

return_code

Possible values are:

CM_OK The call executed successfully.

CM_UNSUCCESSFUL
The receive type is set to CM_RECEIVE_IMMEDIATE, and no data or status information is currently available from the partner program.

CM_DEALLOCATED_NORMAL
The conversation has been deallocated normally. The partner program issued the Deallocate call with the conversation's deallocate type set to one of the following:

- CM_DEALLOCATE_FLUSH
- CM_DEALLOCATE_SYNC_LEVEL with the synchronization level of the conversation specified as CM_NONE

CM_PROGRAM_PARAMETER_CHECK
One of the following has occurred:

- The value specified by *conversation_ID* is not valid
- The value specified by *requested_length* is out of range

If the program receives this return code, the other returned parameters are not valid.

CM_PROGRAM_STATE_CHECK
One of the following has occurred:

- The receive type is set to CM_RECEIVE_AND_WAIT and the conversation state is not Receive, Send, or Send-Pending
- The receive type is set to CM_RECEIVE_IMMEDIATE and the conversation state is not Receive
- The basic conversation is in Send state, the receive type is set to CM_RECEIVE_AND_WAIT, and the program did not finish sending a logical record

If the program receives this return code, the other returned parameters are not valid.

For an explanation of the following return codes, see Appendix B, "Common Return Codes," on page 171.

CM_CONVERSATION_TYPE_MISMATCH
CM_DEALLOCATED_ABEND
CM_DEALLOCATED_ABEND_SVC (basic conversation only)
CM_DEALLOCATED_ABEND_TIMER (basic conversation only)
CM_OPERATION_INCOMPLETE (only if *receive_type* = CM_RECEIVE_AND_WAIT)
CM_OPERATION_NOT_ACCEPTED
CM_PIP_NOT_SPECIFIED_CORRECTLY
CM_PRODUCT_SPECIFIC_ERROR
CM_PROGRAM_ERROR_NO_TRUNC
CM_PROGRAM_ERROR_PURGING
CM_PROGRAM_ERROR_TRUNC (basic conversation only)
CM_RESOURCE_FAILURE_NO_RETRY
CM_RESOURCE_FAILURE_RETRY
CM_SECURITY_NOT_VALID
CM_SYNC_LVL_NOT_SUPPORTED_PGM
CM_SYNC_LVL_NOT_SUPPORTED_LU
CM_TP_NOT_AVAILABLE_NO_RETRY
CM_TP_NOT_AVAILABLE_RETRY
CM_TPN_NOT_RECOGNIZED
CM_SVC_ERROR_NO_TRUNC (basic conversation only)
CM_SVC_ERROR_PURGING (basic conversation only)
CM_SVC_ERROR_TRUNC (basic conversation only)

State When Issued

The conversation can be in Receive, Send, or Send-Pending state.

If *receive_type* is set to CM_RECEIVE_IMMEDIATE, the conversation must be in Receive state.

WINDOWS

If the application successfully issues the Receive call in nonblocking mode, the conversation changes state twice. On the initial return of the call, the conversation changes to Pending-Post state. After CPI-C returns the results of the call processing, the conversation state change is as described below.

Issuing the Call in Send or Send-Pending State

Issuing the Receive call while the conversation is in Send or Send-Pending state causes the local LU to send the information in its send buffer and a send indicator to the partner program. Based on *data_received* and *status_received* parameters, the conversation may change to Receive state for the local program. For more information, see "State Change."

State Change

The new conversation state is determined by the following factors:

- The state the conversation is in when the program issues the call
- The *return_code* parameter
- The *data_received* and *status_received* parameters

Call Issued in Receive State

The state changes shown in Table 18 can occur when the Receive call is issued with the conversation in Receive state and the *return_code* is CM_OK.

Table 18. State Changes When the Receive Call Is Issued in Receive State

data_received	*status_received*	**New state**
CM_DATA_RECEIVED CM_COMPLETE_DATA_RECEIVED CM_INCOMPLETE_DATA_RECEIVED	CM_NO_STATUS_RECEIVED	No change
CM_DATA_RECEIVED CM_COMPLETE_DATA_RECEIVED	CM_SEND_RECEIVED	Send-Pending
CM_NO_DATA_RECEIVED	CM_SEND_RECEIVED	Send

If *return_code* is set to CM_UNSUCCESSFUL, meaning that the *receive_type* is set to CM_RECEIVE_IMMEDIATE and no data is available, there is no state change.

Call Issued in Send State

The state changes shown in Table 19 on page 100 can occur when the Receive call is issued with the conversation in Send state and the *return_code* is CM_OK.

Table 19. State Changes When the Receive Call Is Issued in Send State

data_received	*status_received*	**New state**
CM_DATA_RECEIVED CM_COMPLETE_DATA_RECEIVED CM_INCOMPLETE_DATA_RECEIVED	CM_NO_STATUS_RECEIVED	Receive
CM_DATA_RECEIVED CM_COMPLETE_DATA_RECEIVED	CM_SEND_RECEIVED	Send-Pending
CM_NO_DATA_RECEIVED	CM_SEND_RECEIVED	No change

Call Issued in Send-Pending State

The state changes shown in Table 20 can occur when the Receive call is issued with the conversation in Send-Pending state and the *return_code* is CM_OK.

Table 20. State Changes When the Receive Call Is Issued in Send-Pending State

data_received	*status_received*	**New state**
CM_DATA_RECEIVED CM_COMPLETE_DATA_RECEIVED CM_INCOMPLETE_DATA_RECEIVED	CM_NO_STATUS_RECEIVED	Receive
CM_DATA_RECEIVED CM_COMPLETE_DATA_RECEIVED	CM_SEND_RECEIVED	No change
CM_NO_DATA_RECEIVED	CM_SEND_RECEIVED	Send

Call Issued in Any Allowed State

The following sections summarize state changes that can occur when the Receive call is issued in any allowed state.

Confirmation Processing

The following state changes can occur under the following conditions:

- The *return_code* is CM_OK.
- The *data_received* parameter is set to CM_DATA_RECEIVED, CM_COMPLETE_DATA_RECEIVED, or CM_NO_DATA_RECEIVED.
- The *status_received* parameter indicates a change to a confirm state, as shown in Table 21.

Table 21. State Changes When the Receive Call Is Issued in Any Allowable State

status_received	**New state**
CM_CONFIRM_DEALLOC_RECEIVED	Confirm-Deallocate
CM_CONFIRM_SEND_RECEIVED	Confirm-Send
CM_CONFIRM_RECEIVED	Confirm

Normal Deallocation

If the *return_code* parameter is set to CM_DEALLOCATED_NORMAL, the conversation changes to Reset state.

Abends

The following abend conditions, indicated by the *return_code* parameter, cause the conversation to change to Reset state:

```
CM_CONVERSATION_TYPE_MISMATCH
CM_PIP_NOT_SPECIFIED_CORRECTLY
CM_SECURITY_NOT_VALID
CM_SYNC_LVL_NOT_SUPPORTED_PGM
```

```
CM_SYNC_LVL_NOT_SUPPORTED_LU
CM_TPN_NOT_RECOGNIZED
CM_TP_NOT_AVAILABLE_NO_RETRY
CM_TP_NOT_AVAILABLE_RETRY
CM_DEALLOCATED_ABEND
CM_DEALLOCATED_ABEND_SVC
CM_DEALLOCATED_ABEND_TIMER
CM_SVC_ERROR_TRUNC
CM_RESOURCE_FAILURE_NO_RETRY
CM_RESOURCE_FAILURE_RETRY
```

Errors

The state changes shown in Table 22 can occur when a data transmission error is encountered. (This is indicated by one of the following return codes: CM_PROGRAM_ERROR_PURGING, CM_PROGRAM_ERROR_NO_TRUNC, CM_SVC_ERROR_PURGING, or CM_SVC_ERROR_NO_TRUNC.)

Table 22. State Changes Caused by a Data Transmission Error

return_code	**Old state**	**New state**
CM_PROGRAM_ERROR_PURGING	Receive	No change
CM_PROGRAM_ERROR_NO_TRUNC	Receive	No change
CM_SVC_ERROR_PURGING	Send	Receive
CM_SVC_ERROR_NO_TRUNC	Send-Pending	Receive

Usage Notes

The following sections describe additional usage information for the Receive call.

Truncated Records

If the partner program truncates a logical record, the local program receives notification of the truncation through the *return_code* parameter on the next Receive call.

Setting the Requested_Length Parameter to Zero

If a program issues the Receive call with *requested_length* set to 0 (zero), the call is executed as usual.

However, the *data_received* and *status_received* parameters are not set on the same Receive call. (One exception to this situation is the null record sent over a mapped conversation, described in the next paragraph.)

In a mapped conversation in which data is available from the partner program, the *data_received* parameter is set to CM_INCOMPLETE_DATA_RECEIVED. If a null record is available (*send_length* in the Send_Data call issued by the partner program is set to 0), the *data_received* parameter is set to CM_COMPLETE_DATA_RECEIVED with the *received_length* parameter set to 0 (zero).

In a basic conversation in which data is available and the fill characteristic is set to CM_FILL_LL, the *data_received* parameter is set to CM_INCOMPLETE_DATA_RECEIVED. If the fill characteristic is set to CM_FILL_BUFFER, the *data_received* is set to CM_DATA_RECEIVED.

String Translation

The LU does not automatically perform any conversion between EBCDIC and ASCII on the received string of data before putting it in buffer.

If the remote program sends data in EBCDIC, the local program can use the Convert_Incoming call to convert the received data to ASCII.

WINDOWS

The local program can also use the CSV CONVERT verb to convert the received data to ASCII. Refer to the *Communications Server for AIX CSV Programmer's Guide* for more information.

Release_Local_TP_Name (cmrltp)

AIX, LINUX

The Release_Local_TP_Name call is issued by a program to indicate that it will no longer accept incoming Allocate requests for a TP name. The TP name may have been specified using any of the methods described in "Specifying the Local TP Name" on page 33.

Function Call

```
void cmrltp (
             unsigned char CM_PTR      TP_name,
             CM_INT32 CM_PTR           TP_name_length,
             CM_RETURN_CODE CM_PTR     return_code
            );
```

Function Call for Java CPI-C

```
public native void cmrltp (
                           byte[]          TP_name,
                           CPICLength      TP_name_length,
                           CPICReturnCode  return_code
                          );
```

Supplied Parameters

The supplied parameters are:

TP_name
: This parameter specifies the starting address of the TP name. This must be a TP name that the program has previously specified on a Specify_Local_TP_Name call.

TP_name_length
: This parameter specifies the length of the name (1–64 characters).

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

return_code
: Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
One of the following has occurred:

- The value specified by *TP_name* is not a TP name associated with this program
- The value specified by *TP_name_length* is out of range

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

This call is not associated with a conversation.

State Change

There is no state change.

Usage Notes

If the *return_code* is not CM_OK, the names associated with the program remain unchanged.

If an Accept_Incoming call is outstanding at the time this call is issued, it may accept an incoming Allocate for the name specified on this call. However, subsequent Accept_Conversation or Accept_Incoming calls will not accept incoming Allocates for this name.

If a program releases all its TP names, including the name specified by the APPCTPN environment variable (if any), then it cannot issue any further Accept_Conversation or Accept_Incoming calls unless it first specifies a new local TP name. For more information, see "Specifying the Local TP Name" on page 33.

Request_To_Send (cmrts)

The Request_To_Send call notifies the partner program that the local program wants to send data.

Action of the Partner Program

In response to this request, the partner program can change the conversation to Receive state by issuing one of the following calls:

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

The partner program can also ignore the request to send.

When the Local Program Can Send Data

The conversation state changes to Send for the local program when the local program receives one of the following values through the *status_received* parameter of a subsequent Receive call:

- CM_SEND_RECEIVED
- CM_CONFIRM_SEND_RECEIVED and replies with a Confirmed call

Function Call

```
void cmrts (
            unsigned char CM_PTR       conversation_ID,
            CM_RETURN_CODE CM_PTR      return_code
           );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmrts (
                           byte[]          conversation_ID,
                           CPICReturnCode  return_code
                         );
```

Supplied Parameters

The supplied parameter is:

conversation_ID
This is the identifier for the conversation. The value of this parameter was returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
The value specified by *conversation_ID* is not valid.

CM_PROGRAM_STATE_CHECK
The conversation is not in Receive, Send, Send-Pending, Confirm, Confirm-Send, or Confirm-Deallocate state.

For an explanation of the following return codes, see Appendix B, "Common Return Codes," on page 171.

CM_OPERATION_INCOMPLETE
CM_OPERATION_NOT_ACCEPTED
CM_PRODUCT_SPECIFIC_ERROR

State When Issued

The conversation can be in any of the following states: Receive, Send, Send-Pending, Confirm, Confirm-Send, Confirm-Deallocate, or Pending-Post.

State Change

There is no state change.

Usage Notes

The request-to-send notification is received by the partner program through the *request_to_send_received* parameter of the following calls:

- Confirmed
- Receive
- Send_Data
- Send_Error
- Test_Request_to_Send_Received

Request-to-send notification is sent to the partner program immediately; CPI-C does not wait until the send buffer fills up or is flushed. Consequently, the request-to-send notification may arrive out of sequence. For example, if the local program is in Send state and issues the Prepare_To_Receive call followed by the Request_To_Send call, the partner program, in Receive state, may receive the request-to-send notification before it receives the send notification. For this reason, the request-to-send notification can be reported to a program through the Receive call.

Upon receiving a request-to-send notification, the partner LU retains the notification until the partner program issues a call that returns the parameter *request_to_send_received*. The LU retains only one request-to-send notification per conversation, so the partner program may not be notified of every Request_To_Send issued by the local program.

Send_Data (cmsend)

The Send_Data call puts data in the local LU's send buffer for transmission to the partner program.

The data collected in the local LU's send buffer is transmitted to the partner LU (and partner program) when one of the following occurs:

- The send buffer fills up.
- The local program issues a Flush, Confirm, or Deallocate call or other call that flushes the LU's send buffer. (Some send types, set by the Set_Send_Type call, include flush functionality.)

The data to be sent can be either of the following:

- A complete data record on a mapped conversation. A complete data record is a string of the length specified by the *send_length* parameter.
- A complete logical record, or part of a logical record, on a basic conversation. The length of a complete logical record is determined by the LL value. (One logical record can end and a new one begin in the middle of the string of data to be sent.)

Function Call

```
void cmsend (
            unsigned char CM_PTR              conversation_ID,
            unsigned char CM_PTR              buffer,
```

```
 CM_INT32 CM_PTR                      send_length,
 CM_Request_to_Send_Received CM_PTR   request_to_send_received,
 CM_RETURN_CODE CM_PTR                return_code
);
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmsend (
                            byte[]                          conversation_ID,
                            byte[]                          buffer,
                            CPICLength                      buffer_length,
                            CPICControlInformationReceived  request_to_send_received,
                            CPICReturnCode                  return_code
                          );
```

Supplied Parameters

The supplied parameters are:

conversation_ID
: This is the identifier for the conversation. The value of this parameter was returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

buffer
: This parameter specifies the address of the buffer containing the data to be put in the local LU's send buffer.

send_length
: This is the number of bytes of data to be put in the local LU's send buffer.

  The range for this value is `0`–`32,767`.

  For mapped conversations, if *send_length* is set to `0`, a null data record is sent to the partner program.

  For basic conversations, if *send_length* is set to `0` (zero), no data is sent. The buffer parameter is ignored. However, the other parameters are valid.

Returned Parameters

After the verb executes, CS/AIX returns the following parameters:

request_to_send_received
: This is the request-to-send-received indicator. Possible values are:

  CM_REQ_TO_SEND_RECEIVED
  : The partner program has issued the Request_To_Send call, which requests the local program to change the conversation to Receive state.

  CM_REQ_TO_SEND_NOT_RECEIVED
  : The partner program has not issued the Request_To_Send call.

  This value is not relevant if the *return_code* parameter is set to `CM_PROGRAM_PARAMETER_CHECK` or `CM_PROGRAM_STATE_CHECK`.

return_code
: Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
One of the following has occurred:

- The value specified by *conversation_ID* is not valid
- The value specified by *send_length* is out of range
- This is a basic conversation, and the first 2 bytes of the *buffer* parameter contain a logical record length that is not valid (`0x0000`, `0x0001`, `0x8000`, or `0x8001`)

CM_PROGRAM_STATE_CHECK
One of the following has occurred:

- The conversation state is not Send or Send-Pending.
- The basic conversation is in Send state and the send type is set to `CM_SEND_AND_CONFIRM`, `CM_SEND_AND_DEALLOCATE`, or `CM_SEND_AND_PREP_TO_RECEIVE`. However, the data does not end on a logical record boundary. Send_Data can be issued in the middle of a logical record only when the send type is set to `CM_SEND_AND_DEALLOCATE`, and the deallocate type is set to `CM_DEALLOCATE_ABEND`.

For an explanation of the following return codes, see Appendix B, "Common Return Codes," on page 171.

`CM_CONVERSATION_TYPE_MISMATCH`
`CM_DEALLOCATED_ABEND`
`CM_DEALLOCATED_ABEND_SVC`
`CM_DEALLOCATED_ABEND_TIMER`
`CM_OPERATION_INCOMPLETE`
`CM_OPERATION_NOT_ACCEPTED`
`CM_PIP_NOT_SPECIFIED_CORRECTLY`
`CM_PRODUCT_SPECIFIC_ERROR`
`CM_PROGRAM_ERROR_PURGING`
`CM_RESOURCE_FAILURE_NO_RETRY`
`CM_RESOURCE_FAILURE_RETRY`
`CM_SECURITY_NOT_VALID`
`CM_SVC_ERROR_PURGING`
`CM_SYNC_LVL_NOT_SUPPORTED_PGM`
`CM_SYNC_LVL_NOT_SUPPORTED_LU`
`CM_TP_NOT_AVAILABLE_NO_RETRY`
`CM_TP_NOT_AVAILABLE_RETRY`
`CM_TPN_NOT_RECOGNIZED`

State When Issued

The conversation must be in Send or Send-Pending state when the program issues this call.

State Change

When the *return_code* parameter is set to `CM_OK`, the new conversation state depends on the *send_type* parameter, as shown in Table 23.

Table 23. State Changes for the Send_Data Call

send_type	**New state**
`CM_BUFFER_DATA`	Send
`CM_SEND_AND_FLUSH`	Send

Table 23. State Changes for the Send_Data Call (continued)

send_type	**New state**
CM_SEND_AND_CONFIRM	Send
CM_SEND_AND_PREP_TO_RECEIVE	Receive
CM_SEND_AND_DEALLOCATE	Reset

For a *return_code* value of CM_PROGRAM_ERROR_PURGING or CM_SVC_ERROR_PURGING, the conversation changes to Receive state. For other non-OK values, the conversation changes to Reset state.

Usage Notes

The LU does not automatically perform any conversion between ASCII and EBCDIC on the string of data to be sent.

If the remote program requires data to be sent in EBCDIC, the local program can use the Convert_Outgoing call to convert the data to EBCDIC before sending it.

WINDOWS

The local program can also use the CSV CONVERT verb to convert the data to EBCDIC before sending it. Refer to the *Communications Server for AIX CSV Programmer's Guide* for more information.

Send_Error (cmserr)

The Send_Error call notifies the partner program that the local program has encountered an application-level error. The local program can use the Send_Error for such purposes as informing the partner program of an error encountered in received data, rejecting a confirmation request, or truncating an incomplete logical record it is sending.

The Send_Error call flushes the local LU's send buffer and sends the partner program the contents of the send buffer followed by the error notification.

The error notification is sent to the partner as one of the following *return_code* values:

- CM_PROGRAM_ERROR_TRUNC
- CM_PROGRAM_ERROR_NO_TRUNC
- CM_PROGRAM_ERROR_PURGING

Upon successful execution of this call, the conversation is in Send state for the local program and in Receive state for the partner program.

Function Call

```
void cmserr (
             unsigned char CM_PTR                conversation_ID,
             CM_Request_to_Send_Received CM_PTR  request_to_send_received,
             CM_RETURN_CODE CM_PTR               return_code
            );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmserr (
                           byte[]                          conversation_ID,
                           CPICControlInformationReceived  request_to_send_received,
                           CPICReturnCode                  return_code
                          );
```

Supplied Parameters

The supplied parameter is:

conversation_ID
This is the identifier for the conversation. The value of this parameter was returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

request_to_send_received
This is the request-to-send-received indicator. Possible values are:

CM_REQ_TO_SEND_RECEIVED
The partner program has issued the Request_To_Send call, which requests the local program to change the conversation to Receive state.

CM_REQ_TO_SEND_NOT_RECEIVED
The partner program has not issued the Request_To_Send call.

This value is not relevant if *return_code* is set to CM_PROGRAM_PARAMETER_CHECK or CM_STATE_CHECK.

return_code
The possible return codes vary depending on the conversation state when the call is issued. **Send state**

If the program issues the call with the conversation in Send state the following return codes are possible:

CM_OK The call executed successfully.

For an explanation of the following return codes, see Appendix B, "Common Return Codes," on page 171.
CM_CONVERSATION_TYPE_MISMATCH

CM_DEALLOCATED_ABEND
CM_DEALLOCATED_ABEND_SVC
CM_DEALLOCATED_ABEND_TIMER
CM_OPERATION_INCOMPLETE
CM_OPERATION_NOT_ACCEPTED
CM_PIP_NOT_SPECIFIED_CORRECTLY
CM_PRODUCT_SPECIFIC_ERROR
CM_PROGRAM_ERROR_PURGING
CM_RESOURCE_FAILURE_NO_RETRY
CM_RESOURCE_FAILURE_RETRY
CM_SECURITY_NOT_VALID
CM_SVC_ERROR_PURGING
CM_SYNC_LVL_NOT_SUPPORTED_PGM
CM_SYNC_LVL_NOT_SUPPORTED_LU
CM_TP_NOT_AVAILABLE_NO_RETRY
CM_TP_NOT_AVAILABLE_RETRY
CM_TPN_NOT_RECOGNIZED

Receive state or Pending-Post state

If the call is issued in Receive state or Pending-Post state, the following return codes are possible:

CM_OK Because incoming information is purged when the Send_Error call is issued in Receive state or Pending-Post state, CM_OK is generated instead of the following:
CM_PROGRAM_ERROR_NO_TRUNC
CM_PROGRAM_ERROR_PURGING
CM_SVC_ERROR_NO_TRUNC
CM_SVC_ERROR_PURGING
CM_PROGRAM_ERROR_TRUNC
CM_SVC_ERROR_TRUNC

Common return codes
For an explanation of the following return codes, see Appendix B, "Common Return Codes," on page 171.
CM_OPERATION_INCOMPLETE
CM_OPERATION_NOT_ACCEPTED
CM_PRODUCT_SPECIFIC_ERROR
CM_RESOURCE_FAILURE_NO_RETRY
CM_RESOURCE_FAILURE_RETRY

CM_DEALLOCATED_NORMAL
Because incoming information is purged when the Send_Error call is issued in Receive state or Pending-Post state, CM_DEALLOCATED_NORMAL is generated instead of the following:
CM_CONVERSATION_TYPE_MISMATCH
CM_DEALLOCATED_ABEND
CM_DEALLOCATED_ABEND_SVC
CM_DEALLOCATED_ABEND_TIMER
CM_PIP_NOT_SPECIFIED_CORRECTLY
CM_SECURITY_NOT_VALID
CM_SYNC_LVL_NOT_SUPPORTED_PGM
CM_SYNC_LVL_NOT_SUPPORTED_LU
CM_TPN_NOT_RECOGNIZED
CM_TP_NOT_AVAILABLE_NO_RETRY
CM_TP_NOT_AVAILABLE_RETRY

Send-Pending state

If the call is issued in Send-Pending state, the following return codes are possible:

CM_OK The call executed successfully.

For an explanation of the following return codes, see Appendix B, "Common Return Codes," on page 171.

CM_OPERATION_INCOMPLETE
CM_OPERATION_NOT_ACCEPTED
CM_DEALLOCATED_ABEND
CM_DEALLOCATED_ABEND_SVC
CM_DEALLOCATED_ABEND_TIMER
CM_PRODUCT_SPECIFIC_ERROR
CM_PROGRAM_ERROR_PURGING
CM_RESOURCE_FAILURE_NO_RETRY
CM_RESOURCE_FAILURE_RETRY
CM_SVC_ERROR_PURGING

Confirm, Confirm-Send, or Confirm-Deallocate state

If the call is issued in Confirm, Confirm-Send, or Confirm-Deallocate state, the following return codes are possible:

CM_OK The call executed successfully.

For an explanation of the following return codes, see Appendix B, "Common Return Codes," on page 171.

CM_OPERATION_INCOMPLETE
CM_OPERATION_NOT_ACCEPTED
CM_PRODUCT_SPECIFIC_ERROR
CM_RESOURCE_FAILURE_NO_RETRY
CM_RESOURCE_FAILURE_RETRY

Other states

Issuing the Send_Error call with the conversation in Reset, Initialize, or Initialize-Incoming state is illegal. The following return codes are possible:

CM_OPERATION_NOT_ACCEPTED
See Appendix B, "Common Return Codes," on page 171.

CM_PROGRAM_PARAMETER_CHECK
The value specified by *conversation_ID* is not valid.

CM_PROGRAM_STATE_CHECK
The conversation state is not Send, Receive, Confirm, Confirm-Send, Confirm-Deallocate, or Send-Pending.

State When Issued

The conversation can be in any state except Initialize, Initialize-Incoming, or Reset.

State Change

The new state is determined by the *return_code* parameter. Possible state changes are summarized in Table 24 on page 112.

Table 24. State Changes for the Send_Error Call

return_code	**New state**
CM_OK	Send
CM_CONVERSATION_TYPE_MISMATCH	Reset
CM_PIP_NOT_SPECIFIED_CORRECTLY	
CM_SECURITY_NOT_VALID	
CM_SYNC_LEVEL_NOT_SUPPORTED_PGM	
CM_SYNC_LEVEL_NOT_SUPPORTED_LU	
CM_TPN_NOT_RECOGNIZED	
CM_TP_NOT_AVAILABLE_NO_RETRY	
CM_TP_NOT_AVAILABLE_RETRY	
CM_RESOURCE_FAILURE_RETRY	Reset
CM_RESOURCE_FAILURE_NO_RETRY	
CM_DEALLOCATED_ABEND	Reset
CM_DEALLOCATED_ABEND_SVC	
CM_DEALLOCATED_ABEND_TIMER	
CM_DEALLOCATED_NORMAL	Reset
CM_PROGRAM_ERROR_PURGING	Receive
CM_SVC_ERROR_PURGING	
All others	No change

Usage Notes

The following sections describe additional usage information for the Send_Error call.

Sending Log Data

In basic conversations, the local program can use the Set_Log_Data call to specify error log data to be sent to the partner LU. If the basic conversation's log data length characteristic is greater than 0 (zero), the LU formats the data and stores it in the send buffer.

After the Send_Error call is completed, the log data length is set to 0 (zero) and the log data to null.

Purged Data

If the conversation is in Receive state or Pending-Post state when the program issues the Send_Error call, incoming data is purged by CPI-C. This data includes the following:

- Data sent by the Send_Data call
- Confirmation requests
- Deallocation requests if the conversation's deallocate type is set to CM_DEALLOCATE_CONFIRM or to CM_DEALLOCATE_SYNC_LEVEL with the synchronization level set to CM_CONFIRM

CPI-C does not purge an incoming request-to-send indicator.

Send-Pending State

If the conversation is in Send-Pending state, the local program can issue the Set_Error_Direction call to specify whether the error being reported resulted from the received data or from the processing of the local program after successfully receiving the data.

Set_Conversation_Context (cmsctx)

AIX, LINUX

The Set_Conversation_Context call sets the program's current context to a value that was previously returned on an Extract_Conversation_Context call. This enables the program to start a new conversation using the same context as a previous one.

For more information about conversation contexts, see "Multiple Conversations" on page 12.

Function Call

```
void cmsctx (
             unsigned char CM_PTR      context_ID,
             CM_INT32 CM_PTR           context_ID_length,
             CM_RETURN_CODE CM_PTR     return_code
            );
```

Function Call for Java CPI-C

```
public native void cmsctx (
                             byte[]          context_ID,
                             CPICLength      context_ID_length,
                             CPICReturnCode  return_code
                           );
```

Note: This call is not part of the standard Java CPI-C specification, and is not supported in other Java CPI-C implementations.

Supplied Parameters

The supplied parameters are:

context_ID
This parameter specifies the required context.

context_ID_length
This parameter specifies the length of *context_ID* (1–32 bytes).

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
This return code indicates one of the following cases:

- The value specified by *context_ID* is not the context of any of the program's current conversations, or of its most recent conversation.
- The value specified by *context_ID_length* is not valid.

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

The conversation can be in any state except Reset.

State Change

There is no state change.

Usage Notes

An application uses Set_Conversation_Context in the following situations:

- When it is involved in multiple conversations, and wants to allocate a new conversation using the same context as an existing conversation.
- When a CPI-C call that assigns a new context completes in nonblocking mode. For example, if Accept_Incoming completes immediately with *return_code* CM_OK, the program's current context is set to the context of the new conversation; however, if Accept_Incoming returns CM_OPERATION_INCOMPLETE, a subsequent Wait_For_Conversation that returns the result of Accept_Incoming does not change the program's current context. The program must use Extract_Conversation_Context and Set_Conversation_Context to set the current context to the correct value.

Set_Conversation_Security_Password (cmscsp)

The Set_Conversation_Security_Password call is issued by the invoking program to specify the password required to access the invoked program. This call has an effect on the conversation only if the conversation security type is CM_SECURITY_PROGRAM or CM_SECURITY_PROGRAM_STRONG (AIX or Linux systems), or XC_SECURITY_PROGRAM (Windows systems). It overrides the initial password from the side information specified by the Initialize_Conversation call. This call cannot be issued after the Allocate call has been issued.

Function Call

```
void cmscsp (
             unsigned char CM_PTR      conversation_ID,
             unsigned char CM_PTR      security_password,
             CM_INT32 CM_PTR           security_password_length,
             CM_RETURN_CODE CM_PTR     return_code
            );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmscsp (
                           byte[]          conversation_ID,
                           byte[]          security_password,
                           CPICLength      security_password_length,
                           CPICReturnCode  return_code
                          );
```

Supplied Parameters

The supplied parameters are:

conversation_ID
This is the identifier for the conversation. The value of this parameter is returned by the Initialize_Conversation call.

security_password
This specifies the password required to access the partner program. This parameter is a character string of 1–10 characters(AIX or Linux systems), or 1–8 characters (Windows systems), and is case-sensitive. It must match the password for the user ID configured for the partner program.

The following characters are allowed:

- Uppercase and lowercase letters
- Numerals `0`–`9`
- Special characters `$`, `#`, `@`, and `.` (period)

security_password_length
This specifies the length of *security_password*.

The range for this value is 1–10 characters (AIX or Linux systems), or 1–8 characters (Windows systems). If the *security_password_length* is set to `0` (zero), the *security_password* parameter is ignored; this is equivalent to setting *security_password* to a null string.

Returned Parameters

After the verb executes, CS/AIX returns the following parameters:

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
One of the following has occurred:

- The value specified by *conversation_ID* is not valid
- The value specified by *security_password_length* is out of range

CM_PROGRAM_STATE_CHECK
One of the following has occurred:

- The conversation is not in Initialize state
- The conversation's security type is not set to `CM_SECURITY_PROGRAM` or `CM_SECURITY_PROGRAM_STRONG`

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

The conversation must be in Initialize state.

State Change

There is no state change.

Usage Notes

A user ID is required in addition to the password. This can be obtained from the side information entry specified on the previous Initialize_Conversation call, or the program can specify it using Set_Conversation_Security_User_ID.

A password that is not valid is not detected until the allocation request, generated by the Allocate call, is sent to the partner LU. The error is returned to the invoking program on a subsequent call.

If the return code is not CM_OK, the *security_password* and *security_password_length* conversation characteristics are unchanged.

Set_Conversation_Security_Password (xcscsp)

This function is not available in Java CPI-C.

This call is issued by the invoking program to specify the password required to access the invoked program.

The xcscsp call provides compatibility for applications using the X/Open CPI-C definition. It has been incorporated into IBM CPI-C 2.0 as the call Set_Conversation_Security_Password (cmscsp). Use cmscsp whenever possible to enable greater portability of your program to other platforms.

The parameters on this call are identical to those on the cmscsp call. For more information about cmscsp, see "Set_Conversation_Security_Password (cmscsp)" on page 114.

Set_Conversation_Security_Type (cmscst)

The Set_Conversation_Security_Type call is issued by the invoking program to specify the information the partner LU requires in order to validate access to the invoked program. This call overrides the initial security type from the side information specified by the Initialize_Conversation call. This call cannot be issued after the Allocate has been issued.

Function Call

```
void cmscst (
             unsigned char CM_PTR                   conversation_ID,
             XC_CONVERSATION_SECURITY_TYPE CM_PTR   conversation_security_type,
             CM_RETURN_CODE CM_PTR                  return_code
            );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmscst (
                           byte[]                       conversation_ID,
                           CPICConversationSecurityType conversation_security_type,
                           CPICReturnCode               return_code
                          );
```

Supplied Parameters

The supplied parameters are:

conversation_ID
This is the identifier for the conversation. The value of this parameter is returned by the Initialize_Conversation call.

conversation_security_type
This specifies the information the partner LU requires in order to validate access to the invoked program. Based on the conversation security established for the invoked program during configuration, use one of the following values:

AIX, LINUX

CM_SECURITY_NONE
The invoked program uses no conversation security.

CM_SECURITY_SAME
The invoked program uses conversation security, and is configured to accept an already-verified indicator (as described in "Overview of Conversation Security" on page 12). The user ID from the local program's current context (at the time the Allocate call is issued) will be sent to the invoked program, together with an already-verified indicator. This indicator tells the invoked program not to require the password.

CM_SECURITY_PROGRAM
The invoked program uses conversation security and thus requires a user ID and password. The security information will be taken from the current conversation characteristics (at the time the Allocate call is issued).

CM_SECURITY_PROGRAM_STRONG
As for CM_SECURITY_PROGRAM, except that the local node must not send the password across the network in clear text format. This value can be used only if the remote system supports password substitution.

WINDOWS

XC_SECURITY_NONE
Equivalent to CM_SECURITY_NONE

XC_SECURITY_SAME
Equivalent to CM_SECURITY_SAME

XC_SECURITY_PROGRAM
Equivalent to CM_SECURITY_PROGRAM

Returned Parameters

After the verb executes, CS/AIX returns the following parameters:

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_STATE_CHECK
The conversation is not in Initialize state.

CM_PROGRAM_PARAMETER_CHECK
The value specified by *conversation_ID* or *conversation_security_type* is not valid.

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

The conversation must be in Initialize state.

State Change

There is no state change.

Usage Notes

If the *return_code* is not CM_OK, the *conversation_security_type* is unchanged.

Set_Conversation_Security_Type (xcscst)

This function is not available in Java CPI-C.

This call is issued by the invoking program to specify the information the partner LU requires in order to validate access to the invoked program. This call overrides the initial security type from the side information specified by the Initialize_Conversation call.

The call provides compatibility for applications using the X/Open CPI-C definition. It has been incorporated into IBM CPI-C 2.0 as the call Set_Conversation_Security_Type (cmscst). Use cmscst whenever possible to enable greater portability of your program to other platforms.

The parameters on this call are identical to those on the cmscst call. For more information about cmscst, see "Set_Conversation_Security_Type (cmscst)" on page 116.

Set_Conversation_Security_User_ID (cmscsu)

The Set_Conversation_Security_User_ID call is issued by the invoking program to specify the user ID required to access to the invoked program. It overrides the initial user ID from the side information specified by the Initialize_Conversation call.

This call cannot be issued after the Allocate call has been issued. This call is not valid if the conversation security type is CM_SECURITY_NONE (AIX or Linux systems)or XC_SECURITY_NONE (Windows systems).

Function Call

```
void cmscsu (
             unsigned char CM_PTR      conversation_ID,
             unsigned char CM_PTR      security_user_ID,
             CM_INT32 CM_PTR           security_user_ID_length,
             CM_RETURN_CODE CM_PTR     return_code
            );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmscsu (
                             byte[]          conversation_ID,
                             byte[]          security_user_ID
                             CPICLength      security_user_ID_length,
                             CPICReturnCode  return_code
                           );
```

Supplied Parameters

The supplied parameters are:

conversation_ID
This is the identifier for the conversation. The value of this parameter is returned by the Initialize_Conversation call.

security_user_ID
This specifies the user ID required to access the partner program. This parameter is a character string of 1–10 characters (AIX or Linux systems), or 1–8 characters (Windows systems), and is case-sensitive.

The following characters are allowed:

- Uppercase and lowercase letters
- Numerals `0`–`9`
- Special characters `$`, `#`, `@`, and `.` (period)

security_user_ID_length
This specifies the length of *security_user_ID*. This range for this value is 1–10 characters (AIX or Linux systems), or 1–8 characters (Windows systems). If the length is `0` (zero), the *security_user_ID* parameter is ignored; this is equivalent to setting *security_user_ID* to a null string.

Returned Parameters

After the verb executes, CS/AIX returns the following parameters:

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
One of the following has occurred:

- The value specified by *conversation_ID* is not valid.
- The value specified by *security_user_ID_length* is out of range.

CM_PROGRAM_STATE_CHECK
One of the following has occurred:

- The conversation is not in Initialize state.
- The conversation's security type is set to CM_SECURITY_NONE.

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

The conversation must be in Initialize state.

State Change

There is no state change.

Usage Notes

If the return code is not CM_OK, the *security_user_ID* and *security_user_ID_length* conversation characteristics are unchanged.

A user ID that is not valid is not detected until the allocation request, generated by the Allocate call, is sent to the partner LU. The error is returned to the invoking program on a subsequent call.

Set_Conversation_Security_User_ID (xcscsu)

This function is not available in Java CPI-C.

This call is issued by the invoking program to specify the user ID required to access the invoked program.

The xcscsu call provides compatibility for applications using the X/Open CPI-C definition. It has been incorporated into IBM CPI-C 2.0 as the call Set_Conversation_Security_User_ID (cmscsu). Use cmscsu whenever possible to enable greater portability of your program to other platforms.

The parameters on this call are identical to those on the cmscsu call. For more information about cmscsu, see "Set_Conversation_Security_User_ID (cmscsu)" on page 118.

Set_Conversation_Type (cmsct)

The Set_Conversation_Type call is issued by the invoking program to define a conversation as being mapped or basic. This call overrides the default conversation type established by the Initialize_Conversation call. The default conversation type is CM_MAPPED_CONVERSATION. This call cannot be issued after the Allocate has been issued.

Function Call

```
void cmsct (
           unsigned char CM_PTR          conversation_ID,
           CM_CONVERSATION_TYPE CM_PTR   conversation_type,
           CM_RETURN_CODE CM_PTR         return_code
          );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmsct (
                            byte[]                conversation_ID,
                            CPICConversationType  conversation_type,
                            CPICReturnCode        return_code
                          );
```

Supplied Parameters

The supplied parameters are:

conversation_ID
This is the identifier for the conversation. The value of this parameter is returned by the Initialize_Conversation call.

conversation_type
This parameter specifies the type of conversation to be allocated by the Allocate call. Possible values are:
CM_BASIC_CONVERSATION
CM_MAPPED_CONVERSATION

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_STATE_CHECK
The conversation is not in Initialize state.

CM_PROGRAM_PARAMETER_CHECK
One of the following has occurred:

- The value specified by *conversation_ID* or *conversation_type* is not valid.
- The *conversation_type* parameter specifies a mapped conversation, but the fill characteristic is set to CM_FILL_BUFFER, which is incompatible with mapped conversations. Before changing the conversation type to mapped, you must issue the Set_Fill call to change the fill type to CM_FILL_LL.
- The *conversation_type* parameter specifies a mapped conversation. However, a previous Set_Log_Data call, allowed only in basic conversations, is still in effect.

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, “Common Return Codes,” on page 171.

State When Issued

The conversation must be in Initialize state.

State Change

There is no state change.

Usage Notes

If the return code is not CM_OK, the *conversation_type* conversation characteristic is unchanged.

Set_CPIC_Side_Information (xcmssi)

This function is not available in Java CPI-C.

The Set_CPIC_Side_Information call specifies a side information entry for use by this application. A CPI-C side information entry associates a set of conversation characteristics with a symbolic destination name.

Side information entries are defined in the CS/AIX configuration file. This call specifies an additional entry for use by this application, or overrides the definition in the configuration file (or the application's local definition) if the specified symbolic destination name already exists.

This call is provided for compatibility with X/Open CPI-C and with the Windows CPI-C specification; it is not included in IBM CPI-C 2.0.

Function Call

```
     void xcmssi (
                  unsigned char CM_PTR      key,
                  SIDE_INFO CM_PTR          side_info_entry,
                  CM_INT32 CM_PTR           side_info_entry_length,
                  CM_RETURN_CODE CM_PTR     return_code
                 );

typedef struct side_info_entry
{
  unsigned char     sym_dest_name[8];      /* symbolic destination name      */
  unsigned char     partner_LU_name[17];   /* Fully qualified partner LU name*/
  unsigned char     reserved[3];           /* Reserved                       */
  XC_TP_NAME_TYPE   TP_name_type;          /* TP name type                   */
  unsigned char     TP_name[64];           /* TP name                        */
  unsigned char     mode_name[8];          /* Mode name                      */
  XC_CONVERSATION_SECURITY_TYPE
                    conversation_security_type; /* Conversation security type*/
  unsigned char     security_user_ID[8];   /* User ID                        */
  unsigned char     security_password[8];  /* Password                       */
} SIDE_INFO;
```

Supplied Parameters

The supplied parameters are:

key This parameter is ignored.

side_info_entry
This parameter specifies the contents of a side information entry, as follows. Each field in the structure must be left-justified. Pad fields on the right with spaces as necessary.

side_info_entry.sym_dest_name
Symbolic destination name which identifies the side information entry. The parameter *sym_dest_name* is an 8-byte ASCII character string and can contain any displayable characters.

side_info_entry.partner_LU_name
Fully qualified name of the partner LU. This name is composed of two character strings concatenated by a dot. Each name must can have a maximum length of eight bytes with no embedded spaces; valid characters are uppercase `A`–`Z` and numerals `0`–`9`.

side_info_entry.TP_name_type
The type of the target TP (the valid characters for a TP name are determined by the TP type). Allowed values:

XC_APPLICATION_TP
Application TP. All characters in the TP name must be valid ASCII characters.

XC_SNA_SERVICE_TP
Service TP. The TP name must be specified as an 8–character ASCII string representing the hexadecimal digits of a 4-character name. For example, if the hexadecimal representation of the name is 0x21F0F0F8, set the *tp_name* parameter to the 8–character string "21F0F0F8".

The first character (represented by two bytes) must be a hexadecimal value in the range `0x0`–`0x3F`, excluding `0x0E` and `0x0F`; the remaining characters (each represented by two bytes) must be valid EBCDIC characters.

side_info_entry.TP_name
TP name of the target TP.

Set_CPIC_Side_Information is the only CPI-C call that lets you specify an SNA service TP as the partner program. See the description of the *TP_name_type* parameter above for more information on how to specify the TP name.

side_info_entry.mode_name
Name of the mode used to access the target TP.

For a mapped conversation, the mode name SNASVCMG is reserved for SNA internal use; the Allocate call will fail if you use this name. You are recommended not to use SNASVCMG in a basic conversation, or CPSVCMG (another SNA reserved name) in either type of conversation.

side_info_entry.conversation_security_type
Specifies whether the target TP uses conversation security. Allowed values:

AIX, LINUX

CM_SECURITY_NONE
The target TP does not use conversation security.

CM_SECURITY_PROGRAM
The target TP uses conversation security. The *security_user_ID* and *security_password* parameters specified below will be used to access the target TP.

CM_SECURITY_SAME
The target TP uses conversation security, and can accept an

"already verified" indicator from the local TP. (This indicates that the local TP was itself invoked by another TP, and has verified the security user ID and password supplied by this TP.) The *security_user_ID* parameter specified below will be used to access the target TP; no password is required.

CM_SECURITY_PROGRAM_STRONG
As for CM_SECURITY_PROGRAM, except that the local node must not send the password across the network in clear text format. This value can be used only if the remote system supports password substitution.

WINDOWS

XC_SECURITY_NONE
Equivalent to CM_SECURITY_NONE

XC_SECURITY_SAME
Equivalent to CM_SECURITY_SAME

XC_SECURITY_PROGRAM
Equivalent to CM_SECURITY_PROGRAM

side_info_entry.security_user_ID
User ID used to access the partner TP. This parameter is not required if the *conversation_security_type* parameter is set to CM_SECURITY_NONE.

side_info_entry.security_password
Password used to access the partner TP. This parameter is required only if the *conversation_security_type* parameter is set to CM_SECURITY_PROGRAM or CM_SECURITY_PROGRAM_STRONG.

AIX, LINUX

For compatibility with X/Open CPI-C, this verb only allows eight characters for the user ID and password, although security user IDs can be up to 10 characters. If the partner TP requires a user ID or password of 9 or 10 characters, you must specify it explicitly using the Set_Conversation_Security_User_ID or Set_Conversation_Security_Password call.

side_info_entry_length
This value must always be set to sizeof(SIDE_INFO).

WINDOWS

side_info_entry_length
This value must always be set to 124.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
One of the following has occurred:

- A value specified in the *side_info_entry* structure is not valid
- The first character of the *side_info_entry* contains a space

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

The conversation can be in any state.

State Change

There is no state change.

Usage Notes

This call does not modify the side information held in the configuration file; the change applies only to this application. CS/AIX stores the modified information in memory associated with this operating system process; the change is discarded when the process ends (or when the application issues the Delete_CPIC_Side_Information call to remove the entry). For more details, see "Side Information" on page 30.

If the *return_code* is not CM_OK, the side information is unchanged.

String parameters in the side information that are not valid (for example, specifying a nonexistent partner LU) are not detected until the Allocate call is issued. The error is returned on a call following Allocate.

Set_Deallocate_Type (cmsdt)

The Set_Deallocate_Type call specifies how the conversation is to be deallocated. This call overrides the default deallocate type established by the Initialize_Conversation or Accept_Conversation call. The default deallocate type is CM_DEALLOCATE_SYNC_LEVEL.

The deallocation instructions specified by this call, take effect when the Deallocate call is issued or when the send type is set to CM_SEND_AND_DEALLOCATE and the Send_Data call is issued.

Function Call

```
void cmsdt (
            unsigned char CM_PTR          conversation_ID,
            CM_DEALLOCATE_TYPE CM_PTR     deallocate_type,
            CM_RETURN_CODE CM_PTR         return_code
           );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmsdt (
                            byte[]              conversation_ID,
                            CPICDeallocateType  deallocate_type,
                            CPICReturnCode      return_code
                          );
```

Supplied Parameters

The supplied parameters are:

conversation_ID
This is the identifier for the conversation. The value of this parameter is returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

deallocate_type
This parameter specifies how to perform the deallocation. Possible values are:

CM_DEALLOCATE_ABEND
The conversation is to be deallocated abnormally, unconditionally. A program should specify CM_DEALLOCATE_ABEND when it encounters an error preventing the successful completion of a transaction.

If the conversation is in Send state, CPI-C sends the contents of the local LU's send buffer to the partner program before deallocating the conversation. If the conversation is in Receive state, incoming data may be purged. For a basic conversation in Send state, logical record truncation may occur.

CM_DEALLOCATE_CONFIRM
This value sends the partner program the contents of the local LU's send buffer and a request to confirm the deallocation. The application cannot use this value if the conversation's synchronization level is CM_NONE.

This request for deallocation confirmation is sent by the Deallocate call or by the Send_Data call with the send type set to CM_SEND_AND_DEALLOCATE. The conversation is deallocated normally when the partner program issues the Confirmed call, responding to the confirmation request.

CM_DEALLOCATE_FLUSH
This value sends the contents of the local LU's send buffer to the partner program before deallocating the conversation normally.

CM_DEALLOCATE_SYNC_LEVEL
This value uses the conversation's synchronization level to determine how to deallocate the conversation. A default synchronization level is established by the Initialize_Conversation call and can be overridden by the Set_Sync_Level call.

If the synchronization level of the conversation is set to the default, CM_NONE, the contents of the local LU's send buffer are sent to the partner program and the conversation is deallocated normally.

If the synchronization level of the conversation is CM_CONFIRM, the contents of the local LU's send buffer and a request to confirm the deallocation are sent to the partner program. This request for deallocation confirmation is sent by Deallocate call, or by the Send_Data call with the send type set to CM_SEND_AND_DEALLOCATE. The conversation is deallocated normally when the partner program issues the Confirmed call, responding to the confirmation request.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
One of the following has occurred:

- The value specified by *conversation_ID* or *deallocate_type* is not valid
- The *deallocate_type* parameter specifies CM_DEALLOCATE_CONFIRM, but the conversation's synchronization level is set to CM_NONE

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

The conversation can be in any state except Reset.

State Change

There is no state change.

Usage Notes

If the *return_code* is not CM_OK, the *deallocate_type* conversation characteristic is unchanged.

You can set *deallocate_type* to CM_FLUSH if the synchronization level of the conversation is set to CM_NONE or CM_CONFIRM.

The value CM_DEALLOCATE_FLUSH is equivalent to CM_DEALLOCATE_SYNC_LEVEL with the conversation's synchronization level set to CM_NONE.

The value CM_DEALLOCATE_CONFIRM is equivalent to CM_DEALLOCATE_SYNC_LEVEL with the conversation's synchronization level set to CM_CONFIRM.

Set_Error_Direction (cmsed)

The Set_Error_Direction call specifies whether a program detected an error while receiving data or while preparing to send data. This call overrides the default error direction established by the Initialize_Conversation or Accept_Conversation call. The default error direction is CM_RECEIVE_ERROR.

Error direction is relevant only when a program issues the Send_Error call in Send-Pending state immediately after issuing the Receive call and receiving data (*data_received* is a value other than CM_NO_DATA_RECEIVED) and a send indicator (*status_received* = CM_SEND_RECEIVED).

Function Call

```
void cmsed (
            unsigned char CM_PTR          conversation_ID,
            CM_ERROR_DIRECTION CM_PTR     error_direction,
            CM_RETURN_CODE CM_PTR         return_code
           );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmsed (
                           byte[]              conversation_ID,
                           CPICErrorDirection  error_direction,
                           CPICReturnCode      return_code
                         );
```

Supplied Parameters

The supplied parameters are:

conversation_ID
: This is the identifier for the conversation. The value of this parameter is returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

error_direction
: This parameter specifies the direction in which data was flowing when the program encountered an error. Possible values are:

  CM_RECEIVE_ERROR
  : An error occurred in the data received from the partner program.

  CM_SEND_ERROR
  : An error occurred while the local program prepared to send data to the partner program.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

return_code
: Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
The value specified by *conversation_ID* or *error_direction* is not valid.

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

The conversation can be in any state except Reset.

State Change

There is no state change.

Usage Notes

If the *return_code* is not CM_OK, the *error_direction* conversation characteristic is unchanged.

When the conversation is in Send-Pending state, the program issues the Send_Error call if it detects errors in the received data or if an error occurred while the local program prepared to send data. The program must supply the error direction information using the Set_Error_Direction call before issuing the Send_Error call because the LU cannot tell which kind of error occurred (receive or send). The new error direction remains in effect until a subsequent Set_Error_Direction changes it.

When the Send_Error call is issued, the partner program receives one of the following return codes:

- CM_PROGRAM_ERROR_PURGING if *error_direction* is set to CM_RECEIVE_ERROR
- CM_PROGRAM_ERROR_NO_TRUNC if *error_direction* is set to CM_SEND_ERROR

Set_Fill (cmsf)

The Set_Fill call specifies whether programs will receive data in the form of logical records or as a specified length of data. This call is allowed only in basic conversations. It overrides the default fill established by the Initialize_Conversation or Accept_Conversation call. The default fill is CM_FILL_LL.

The fill value affects all subsequent Receive calls. It can be changed by issuing the Set_Fill call again.

Function Call

```
void cmsf  (
            unsigned char CM_PTR       conversation_ID,
            CM_FILL CM_PTR             fill,
            CM_RETURN_CODE CM_PTR      return_code
           );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmsf (
                           byte[]          conversation_ID,
                           CPICFill        fill,
                           CPICReturnCode  return_code
                         );
```

Supplied Parameters

The supplied parameters are:

conversation_ID
This is the identifier for the conversation. The value of this parameter is returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

fill This parameter specifies the form in which programs will receive data. Possible values are:

CM_FILL_BUFFER
The local program receives data until the number of bytes specified by the *requested_length* parameter of the Receive call is reached, or until the end of the data. Data is received without regard for the logical-record format.

CM_FILL_LL
Data is received in logical-record format. The data received can be any of the following:

- A complete logical record
- A portion of a logical record equal to the *requested_length* parameter of the Receive call
- The end of a logical record

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
One of the following has occurred:

- The value specified by the *conversation_ID* or *fill* parameter is not valid
- The current conversation is mapped. The *fill* parameter does not apply to mapped conversations

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

The conversation can be in any state except Reset.

State Change

There is no state change.

Usage Notes

If the *return_code* is not CM_OK, the *fill* conversation characteristic is unchanged.

Set_Local_LU_Name (cmslln)

The Set_Local_LU_Name call is issued by the invoking program to specify the local LU for a conversation. This call overrides the system-defined Local LU derived from the side information when Initialize_Conversation was issued, and any Local LU specified by the APPCLLU environment variable. This call cannot be issued after the Allocate has been issued. Issuing this call has no effect on the side information itself.

This call is not part of the standard CPI-C specification, and may not be available in other implementations. In particular, it is not supported in other Java CPI-C implementations.

Function Call

```
void cmslln   (
               unsigned char CM_PTR         Conversation_ID,
               unsigned char CM_PTR         lu_alias,
               CM_INT32 CM_PTR              lu_alias_length,
               CM_RETURN_CODE CM_PTR        return_code
              );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmslln (
                             byte[]          conversation_ID,
                             byte[]          lu_alias,
                             CPICLength      lu_alias_length,
                             CPICReturnCode  return_code
                           );
```

Supplied Parameters

The supplied parameters are:

conversation_ID
: This is the identifier for the conversation. The value of this parameter is returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

lu_alias
: This parameter specifies the starting address of the LU alias. The LU alias can contain up to eight ASCII characters.

lu_alias_length
: This parameter specifies the length of the LU alias. The range for this value is 0–8 bytes. If *lu_alias_length* is 0 (zero), the LU alias is set to all zeros.

Returned Parameters

After the verb executes, CS/AIX returns the following parameters:

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_STATE_CHECK
The conversation is not in Initialize state.

CM_PROGRAM_PARAMETER_CHECK
One of the following has occurred:

- The value specified by *conversation_ID* is not valid
- The value specified by *lu_alias_length* is out of range (greater than 8 or less than 0).

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

The conversation must be in Initialize state.

State Change

There is no state change.

Usage Notes

If the *return_code* is not CM_OK, the *lu_alias* conversation characteristic is unchanged.

Specifying a value for *lu_alias* that is not valid (a name that is not permitted by the configuration file) is not detected until the Allocate call is issued.

Set_Log_Data (cmsld)

The Set_Log_Data call specifies a log message (log data) and its length to be sent to the partner LU. This call is allowed only in basic conversations. It overrides the default log data, which is null, and the default log data length, which is 0 (zero).

Function Call

```
void 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
           );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmsld (
                           byte[]          conversation_ID,
                           byte[]          log_data,
                           CPICLength      log_data_length,
                           CPICReturnCode  return_code
                          );
```

Supplied Parameters

The supplied parameters are:

conversation_ID
: This is the identifier for the conversation. The value of this parameter is returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

log_data
: Address of data buffer containing error information. This data is sent to the local error log and to the partner LU.

  This parameter is used by the Send_Error call if *log_data_length* is greater than `0` (zero).

  The program must format the error data as a General Data Stream (GDS) error log variable. For further information, refer to the IBM publication *IBM Systems Network Architecture: LU 6.2 Reference: Peer Protocols*.

log_data_length
: This parameter specifies the length of the log data.

  The range for this value is `0`–`512` bytes.

  A length of `0` (zero) indicates that there is no log data. The *log_data* parameter is ignored, and the *log_data* conversation characteristic is set to a null string.

Returned Parameters

After the verb executes, CS/AIX returns the following parameters:

return_code
: Possible values are:

  CM_OK The call executed successfully.

  CM_PROGRAM_PARAMETER_CHECK
  : One of the following has occurred:

    - The value specified by *conversation_ID* is not valid.
    - The conversation type is set to mapped.
    - The value specified by *log_data_length* is out of range (greater than `512` or less than `0`).

  CM_PRODUCT_SPECIFIC_ERROR
  : For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

The conversation can be in any state except Reset.

State Change

There is no state change.

Usage Notes

If the *return_code* is not CM_OK, the *log_data* and *log_data_length* conversation characteristics are unchanged.

The log data specified by the Set_Log_Data call is sent to the partner LU when the local program issues one of the following calls:

- Send_Error
- Deallocate with the conversation's *deallocate_type* set to CM_DEALLOCATE_ABEND
- Send_Data with the conversation's *send_type* set to CM_SEND_AND_DEALLOCATE and the *deallocate_type* set to CM_DEALLOCATE_ABEND

After sending the log data to the partner LU, the local LU resets the log data to null and the log data length to 0 (zero).

CPI-C automatically converts the log data from ASCII to EBCDIC as required.

Set_Mode_Name (cmsmn)

The Set_Mode_Name call is issued by the invoking program to specify the mode name for a conversation. This call overrides the system-defined mode name derived from the side information when the Initialize_Conversation call was issued. This call cannot be issued after the Allocate has been issued. Issuing this call has no effect on the side information itself.

Function Call

```
void 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
           );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmsmn (
                           byte[]          conversation_ID,
                           byte[]          mode_name,
                           CPICLength      mode_name_length,
                           CPICReturnCode  return_code
                          );
```

Supplied Parameters

The supplied parameters are:

conversation_ID
: This is the identifier for the conversation. The value of this parameter is returned by the Initialize_Conversation call.

mode_name
: This parameter specifies the starting address of the mode name (the name

of a set of networking characteristics defined during configuration). The mode name can contain up to eight ASCII characters. The following characters are allowed:

- Uppercase letters
- Numerals 0–9

The first character of the name must be a letter, or can be # for one of the SNA-defined modes such as #INTER. For information about SNA-defined modes, see the *Communications Server for AIX Administration Guide*.

The value of *mode_name* must match the name of a mode associated with the partner LU during configuration.

For a mapped conversation, the mode name SNASVCMG is reserved for SNA internal use; the Allocate call will fail if you use this name. You are recommended not to use SNASVCMG in a basic conversation, or CPSVCMG (another SNA reserved name) in either type of conversation.

mode_name_length
This parameter specifies the length of the mode name.

The range for this value is 0–8 bytes.

If *mode_name_length* is set to 0 (zero), the Set_Mode_Name call is ignored.

Returned Parameters

After the verb executes, CS/AIX returns the following parameters:

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_STATE_CHECK
The conversation is not in Initialize state.

CM_PROGRAM_PARAMETER_CHECK
One of the following has occurred:

- The value specified by *conversation_ID* is not valid
- The value specified by *mode_name_length* is out of range (greater than 8 or less than 0).

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

The conversation must be in Initialize state.

State Change

There is no state change.

Usage Notes

If the *return_code* is not CM_OK, the *mode_name* conversation characteristic is unchanged.

Specifying a value for *mode_name* that is not valid (a name that is not permitted by the configuration file) is not detected until the Allocate call is issued.

Set_Partner_LU_Name (cmspln)

The Set_Partner_LU_Name call is issued by the invoking program to specify the partner LU name. This call overrides the partner LU name derived from the side information when the Initialize_Conversation call was issued. This call cannot be issued after the Allocate has been issued. Issuing this call has no effect on the side information itself.

Function Call

```
void 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
            );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmspln (
                             byte[]          conversation_ID,
                             byte[]          partner_LU_name,
                             CPICLength      partner_LU_name_length,
                             CPICReturnCode  return_code
                           );
```

Supplied Parameters

The supplied parameters are:

conversation_ID
This is the identifier for the conversation. The value of this parameter is returned by the Initialize_Conversation call.

partner_LU_name
This parameter specifies the starting address of the partner LU name. The following characters are allowed:

- Uppercase letters
- Numerals 0–9

The partner LU name can be either of the following:

- An alias consisting of 1–8 ASCII characters.
- A fully qualified network name consisting of 2–17 ASCII characters. A period (.) separates the network ID (which can be 0–8 characters) from the network LU name (which can be 1–8 characters). If the network ID is zero characters long, the period is still required.

If the partner LU is specified by its alias, this must match the alias defined for a partner LU in the CS/AIX configuration.

partner_LU_name_length
This parameter specifies the length of the partner LU name.

The range for this value is 1–17.

Returned Parameters

After the verb executes, CS/AIX returns the following parameters:

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_STATE_CHECK
The conversation is not in Initialize state.

CM_PROGRAM_PARAMETER_CHECK
One of the following has occurred:

- The value specified by *conversation_ID* is not valid
- The value specified by *partner_LU_name_length* is out of range

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

The conversation must be in Initialize state.

State Change

There is no state change.

Usage Notes

If the *return_code* is not CM_OK, the *partner_LU_name* conversation characteristic is unchanged.

Specifying a value for *partner_LU_name* that is not valid (a name not permitted by the configuration) is not detected until the Allocate call is issued.

Set_Prepare_To_Receive_Type (cmsptr)

The Set_Prepare_To_Receive_Type call specifies how the subsequent Prepare_To_Receive calls will be executed. It overrides the default prepare-to-receive processing established by the Initialize_Conversation or Accept_Conversation call. By default, the prepare-to-receive processing is based on the synchronization level of the conversation.

The prepare to receive type affects all subsequent Prepare_To_Receive calls. It can be changed by issuing the Set_Prepare_To_Receive_Type call again.

Function Call

```
void 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
           );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmsptr (
                             byte[]                    conversation_ID,
                             CPICPrepareToReceiveType  prepare_to_receive_type,
                             CPICReturnCode            return_code
                           );
```

Supplied Parameters

The supplied parameters are:

conversation_ID
This is the identifier for the conversation. The value of this parameter is returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

prepare_to_receive_type
This parameter specifies how subsequent Prepare_To_Receive calls will be executed. Possible values are:

CM_PREP_TO_RECEIVE_CONFIRM
This value sends the partner program the contents of the LU's send buffer and a confirmation request. Upon receipt of confirmation, the conversation changes to Receive state.

CM_PREP_TO_RECEIVE_FLUSH
This value sends the partner program the contents of the local LU's send buffer and changes the conversation to Receive state.

CM_PREP_TO_RECEIVE_SYNC_LEVEL
This value uses the conversation's synchronization level to determine prepare-to-receive processing. A default synchronization level is established by the Initialize_Conversation call and can be overridden by the Set_Sync_Level call.

If the synchronization level of the conversation is set to the default, CM_NONE, the contents of the local LU's send buffer are sent to the partner program and the conversation changes to Receive state.

If the synchronization level of the conversation is CM_CONFIRM, the contents of the local LU's send buffer and a request for confirmation are sent to the partner program. The conversation changes to Receive state when the partner program issues the Confirmed call, responding to the confirmation request.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
One of the following has occurred:

- The value specified by the *prepare_to_receive_type* or *conversation_ID* parameter is not valid

- The *prepare_to_receive_type* parameter is set to CM_PREP_TO_RECEIVE_CONFIRM, but the conversation's synchronization level is set to CM_NONE

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

The conversation can be in any state except Reset.

State Change

There is no state change.

Usage Notes

If the *return_code* is not CM_OK, the *prepare_to_receive_type* conversation characteristic is unchanged.

Set_Processing_Mode (cmspm)

This function is not available in Java CPI-C. Java CPI-C functions always operate in blocking mode; that is, the function does not return control to the application until the requested processing has completed.

The Set_Processing_Mode call specifies whether subsequent CPI-C calls will return when the required operation is complete (blocking mode), or return immediately even if the operation is not complete (nonblocking mode). The default processing mode, established by the Initialize_Conversation or Accept_Conversation call, is CM_BLOCKING (blocking mode).

AIX, LINUX

If the conversation's processing mode is nonblocking, CPI-C calls issued on this conversation can return immediately with a return code of CM_OPERATION_INCOMPLETE to indicate that the requested operation has not been completed. The application can then perform other processing not related to this conversation, or can issue any of the following calls:

- Check_For_Completion, to determine whether any outstanding call (on this or any other conversation) has completed
- Wait_For_Conversation, to wait for this call to complete
- Cancel_Conversation, to cancel the outstanding call and deallocate the conversation

WINDOWS

A Windows application can use the Wait_For_Conversation call, as described previously. However, the recommended method for handling nonblocking calls is to use Specify_Windows_Handle. This function, which must be issued before any nonblocking calls, specifies a Windows handle to which CPI-C sends a message when the call processing has completed. The application checks the results of the

call when it receives this message, and does not use Wait_For_Conversation. Check_For_Completion, described previously for AIX or Linux systems, is not supported on Windows systems.

If the outstanding call is a Receive call, a Windows application can issue the Request_To_Send, Send_Error, Test_Request_to_Send_Received, or Deallocate calls in addition to those listed previously. For more information, see "Receive (cmrcv)" on page 94.

The processing mode affects all subsequent CPI-C calls. It can be changed by issuing the Set_Processing_Mode call again.

Function Call

```
void cmspm (
            unsigned char CM_PTR       conversation_ID,
            CM_INT32 CM_PTR            processing_mode,
            CM_RETURN_CODE CM_PTR      return_code
           );
```

Supplied Parameters

The supplied parameters are:

conversation_ID
This is the identifier for the conversation. The value of this parameter is returned by the Initialize_Conversation or Accept_Conversation call.

processing_mode
This parameter specifies whether subsequent CPI-C calls will be executed in blocking or nonblocking mode. Possible values are:

CM_BLOCKING
Subsequent CPI-C calls will not return until the operation is complete.

CM_NON_BLOCKING
Subsequent CPI-C calls will return immediately after the operation is initiated, whether or not it has completed.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
The value specified by the *processing_mode* or *conversation_ID* parameter is not valid.

For an explanation of the following return codes, see Appendix B, "Common Return Codes," on page 171.

CM_OPERATION_NOT_ACCEPTED
CM_PRODUCT_SPECIFIC_ERROR

State When Issued

The conversation can be in any state except Reset.

State Change

There is no state change.

Usage Notes

If the *return_code* is not CM_OK, the *processing_mode* conversation characteristic is unchanged.

Set_Receive_Type (cmsrt)

The Set_Receive_Type call specifies how the program will receive data on subsequent Receive calls. It overrides the default receive type established by the Initialize_Conversation or Accept_Conversation call. By default, the program waits for data to arrive if it is not available when the Receive call is issued.

The receive type value affects all subsequent Receive calls. It can be changed by issuing the Set_Receive_Type call again.

Function Call

```
void cmsrt (
            unsigned char CM_PTR       conversation_ID,
            CM_RECEIVE_TYPE CM_PTR     receive_type,
            CM_RETURN_CODE CM_PTR      return_code
           );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmsrt (
                            byte[]           conversation_ID,
                            CPICReceiveType  receive_type,
                            CPICReturnCode   return_code
                          );
```

Supplied Parameters

The supplied parameters are:

conversation_ID
: This is the identifier for the conversation. The value of this parameter is returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

receive_type
: This parameter specifies how data is to be received by the program on the subsequent Receive calls. Possible values are:

  CM_RECEIVE_AND_WAIT
  : The local program receives any data that is currently available from the partner program. If no data is currently available, the local program waits for data to arrive.

CM_RECEIVE_IMMEDIATE
The local program receives any data currently available from the partner program. If no data is available, the local program does not wait.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
The value specified by *conversation_ID* or *receive_type* is not valid.

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

The conversation can be in any state except Reset.

State Change

There is no state change.

Usage Notes

If the *return_code* is not CM_OK, the *receive_type* conversation characteristic is unchanged.

Set_Return_Control (cmsrc)

The Set_Return_Control call is issued by the invoking program to specify whether the Allocate call returns immediately if a session is not available, or waits for a session to be allocated.

This call overrides the default return control established by the Initialize_Conversation call. By default, CPI-C waits for the session to be allocated. This call cannot be issued after the Allocate call has been issued.

For further information about sessions, see Chapter 2, "Writing CPI-C Applications," on page 19.

Function Call

```
void cmsrc (
           unsigned char CM_PTR         conversation_ID,
           CM_RETURN_CONTROL CM_PTR     return_control,
           CM_RETURN_CODE CM_PTR        return_code
          );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmsrc (
                            byte[]             conversation_ID,
                            CPICReturnControl  return_control,
                            CPICReturnCode     return_code
                          );
```

Supplied Parameters

The supplied parameters are:

conversation_ID
: This is the identifier for the conversation. The value of this parameter is returned by the Initialize_Conversation call.

return_control
: This parameter specifies when the local LU, acting on the Allocate call, should return control to the local program. The following are allowed values:

CM_IMMEDIATE
: The LU allocates a contention winner session, if one is immediately available, and returns control to the program.

CM_WHEN_SESSION_ALLOCATED
: The LU does not return control to the program until it allocates a session or encounters certain errors. If a session is not available, the program waits for one. (If the session limit is `0`, the LU returns control immediately.)

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

return_code
: Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_STATE_CHECK
: The conversation is not in Initialize state.

CM_PROGRAM_PARAMETER_CHECK
: The value specified by *conversation_ID* or *return_control* is not valid.

CM_PRODUCT_SPECIFIC_ERROR
: For an explanation of this return code, see Appendix B, “Common Return Codes,” on page 171.

State When Issued

The conversation must be in Initialize state.

State Change

There is no state change.

Usage Notes

If the *return_code* is not CM_OK, the *return_control* conversation characteristic is unchanged.

If the LU is unable to allocate a session, the notification is returned on the Allocate call.

Set_Send_Type (cmsst)

The Set_Send_Type call specifies how data will be sent by the next Send_Data call. It overrides the default send type established by the Initialize_Conversation or Accept_Conversation call. The default send type is CM_BUFFER_DATA, indicating that data only (and no control information) is to be sent.

The send type value affects all subsequent Send_Data calls. It can be changed by issuing the Set_Send_Type call again.

Function Call

```
void cmsst (
            unsigned char CM_PTR       conversation_ID,
            CM_SEND_TYPE CM_PTR        send_type,
            CM_RETURN_CODE CM_PTR      return_code
           );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmsst (
                            byte[]          conversation_ID,
                            CPICSendType    send_type,
                            CPICReturnCode  return_code
                          );
```

Supplied Parameters

The supplied parameters are:

conversation_ID
: This is the identifier for the conversation. The value of this parameter is returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

send_type
: This parameter specifies how data is to be sent by subsequent Send_Data calls. Possible values are:

  CM_BUFFER_DATA
  : The data pointed to by the Send_Data call is stored in a buffer until the buffer fills up or is flushed.

  CM_SEND_AND_FLUSH
  : The data pointed to by the Send_Data call is to be sent immediately. This is equivalent to Send_Data, with the *send_type* set to CM_BUFFER_DATA, followed by Flush.

CM_SEND_AND_CONFIRM
The data is to be sent immediately with a request for confirmation. This is equivalent to Send_Data, with the *send_type* set to CM_BUFFER_DATA, followed by Confirm.

CM_SEND_AND_PREP_TO_RECEIVE
The data is to be sent immediately along with notification to the partner program that the conversation state for the sending program is changing to Receive. This is equivalent to Send_Data, with the *send_type* set to CM_BUFFER_DATA, followed by Prepare_To_Receive.

CM_SEND_AND_DEALLOCATE
The data is to be sent immediately along with deallocation notification. This is equivalent to Send_Data, with the *send_type* set to CM_BUFFER_DATA, followed by Deallocate.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
One of the following has occurred:

- The value specified by *conversation_ID* or *send_type* is not valid
- The *send_type* parameter is set to CM_SEND_AND_CONFIRM, but the conversation's synchronization level is set to CM_NONE

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

The conversation can be in any state except Reset.

State Change

There is no state change.

Usage Notes

If the *return_code* is not CM_OK, the *send_type* conversation characteristic is unchanged.

Using *send_type* values other than CM_BUFFER_DATA enables you to reduce the number of calls issued, because with these values a Send_Data call can include the function of another CPI-C call.

Set_Sync_Level (cmssl)

The Set_Sync_Level call is issued by the invoking program to specify the synchronization level of the conversation. The synchronization level determines whether the programs synchronize their processing through the Confirm and Confirmed calls.

This call overrides the synchronization level established by the Initialize_Conversation call. The default synchronization level is CM_NONE, indicating no synchronization. This call cannot be issued after the Allocate call has been issued.

Function Call

```
void cmssl (
            unsigned char CM_PTR       conversation_ID,
            CM_SYNC_LEVEL CM_PTR       sync_level,
            CM_RETURN_CODE CM_PTR      return_code
           );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmssl (
                            byte[]          conversation_ID,
                            CPICSyncLevel   sync_level,
                            CPICReturnCode  return_code
                          );
```

Supplied Parameters

The supplied parameters are:

conversation_ID
This is the identifier for the conversation. The value of this parameter is returned by the Initialize_Conversation call.

sync_level
This parameter specifies the synchronization level of the conversation. Possible values are:

CM_NONE
The programs will not perform confirmation processing.

CM_CONFIRM
The programs can perform confirmation processing.

A third level, sync point, is provided by some CPI-C implementations, but is not supported by CS/AIX CPI-C.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_STATE_CHECK
The conversation is not in Initialize state.

CM_PROGRAM_PARAMETER_CHECK
One of the following has occurred:

- The value specified by *conversation_ID* or *sync_level* is not valid
- The *sync_level* parameter specifies CM_NONE but one of the following has occurred:
  - The *send_type* parameter is set to CM_SEND_AND_CONFIRM
  - The *prepare_to_receive_type* parameter is set to CM_PREP_TO_RECEIVE_CONFIRM
  - The *deallocate_type* parameter is set to CM_DEALLOCATE_CONFIRM

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

The conversation must be in Initialize state.

State Change

There is no state change.

Usage Notes

If the *return_code* is not CM_OK, the *sync_level* conversation characteristic is unchanged.

Set_TP_Name (cmstpn)

The Set_TP_Name call is issued by the invoking program to specify the partner program name. This call overrides the partner program name derived from the side information when the Initialize_Conversation call was issued. This call cannot be issued after the Allocate call has been issued. Issuing this call has no effect on the side information itself.

This call functions differently from Specify_Local_TP_Name. Set_TP_Name is issued by the invoking program, to specify the name of the program it wants to allocate a conversation with; Specify_Local_TP_Name is issued by the invoked program, to specify a name for which it will accept incoming Allocate requests.

Function Call

```
void 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
            );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmstpn (
                             byte[]          conversation_ID,
                             byte[]          TP_name,
                             CPICLength      TP_name_length,
                             CPICReturnCode  return_code
                           );
```

Supplied Parameters

The supplied parameters are:

conversation_ID
This is the identifier for the conversation. The value of this parameter is returned by the Initialize_Conversation call.

TP_name
This parameter specifies the starting address of the partner program name. The program name can contain up to 64 characters. The following characters are allowed:

- Uppercase and lowercase letters
- Numerals 0–9 and . (period)
- The following special characters: < > ( ) + - & *; / , % _ ? : ' = " (valid only if the partner program is a CPI-C program) $ # @ (valid only if the partner program is an APPC program)

You cannot use the Set_TP_Name call to specify the name of an SNA service TP, which contains characters that are not allowed for this call. You can, however, use the Set_CPIC_Side_Information call to do this.

Double-byte character sets, such as Kanji, are not supported.

TP_name_length
This parameter specifies the length of the partner program name.

The range for this value is 1–64.

Returned Parameters

After the verb executes, CS/AIX returns the following parameters:

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_STATE_CHECK
The conversation is not in Initialize state.

CM_PROGRAM_PARAMETER_CHECK
One of the following has occurred:

- The value specified by *conversation_ID* is not valid
- The value specified by *TP_name_length* is out of range

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

The conversation must be in Initialize state.

State Change

There is no state change.

Usage Notes

If the *return_code* is not CM_OK, the *TP_name* conversation characteristic is unchanged.

Specify_Local_TP_Name (cmsltp)

The Specify_Local_TP_Name call is issued by a CPI-C application to specify a local TP name for which it will accept incoming Allocate requests.

Instead of using this call, you can set the local TP name in other ways such as by using the APPCTPN environment variable. For more information about setting the local TP name, see "Specifying the Local TP Name" on page 33. The Specify_Local_TP_Name call is required only when a single application wishes to accept incoming Allocates for more than one local TP name; it can use APPCTPN for one name, but must use this call to specify additional names. (After issuing the Accept_Conversation or Accept_Incoming call to accept an incoming Allocate request, it can use Extract_TP_Name to determine which of the names was specified by the partner application.)

This call functions differently from Set_TP_Name. Set_TP_Name is issued by the invoking program, to specify the name of the program it wants to allocate a conversation with; Specify_Local_TP_Name is issued by the invoked program, to specify a name for which it will accept incoming Allocate requests.

Function Call

```
void cmsltp (
             unsigned char CM_PTR      TP_name,
             CM_INT32 CM_PTR           TP_name_length,
             CM_RETURN_CODE CM_PTR     return_code
            );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmsltp (
                             byte[]          TP_name,
                             CPICLength      TP_name_length,
                             CPICReturnCode  return_code
                           );
```

Supplied Parameters

The supplied parameters are:

TP_name
: This parameter specifies the starting address of the TP name. The name can contain up to 64 characters. The following characters are allowed:

  - Uppercase and lowercase letters
  - Numerals 0–9
  - The special characters: . < > ( ) + - & *; / , % _ ? : ' = "

  You cannot use the Specify_Local_TP_Name call to specify the name of an SNA service TP, which contains characters that are not allowed for this call.

  Double-byte character sets, such as Kanji, are not supported.

TP_name_length
: This parameter specifies the length of the name.

  The range for this value is 1–64.

Returned Parameters

After the verb executes, CS/AIX returns the following parameters:

return_code
: Possible values are:

  CM_OK The call executed successfully.

  CM_PROGRAM_PARAMETER_CHECK
  : One of the following has occurred:

    - The value specified by *TP_name* is a reserved name, or contains one or more characters that are not valid.
    - The value specified by *TP_name_length* is out of range.

  CM_PRODUCT_SPECIFIC_ERROR
  : For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

This call is not associated with a conversation.

State Change

There is no state change.

Usage Notes

If the *return_code* is not CM_OK, the TP names for which this program will accept incoming Allocate requests are unchanged.

If an Accept_Incoming call is outstanding at the time this call is issued, it will not accept an incoming Allocate for the name specified on this call. However, subsequent Accept_Conversation or Accept_Incoming calls will accept incoming Allocates for this name.

Specify_Windows_Handle (xchwnd)

WINDOWS

The Specify_Windows_Handle call is issued by a CPI-C application to specify a Windows handle to which CPI-C will send a message each time a nonblocking CPI-C function completes. This provides an alternative mechanism to using Wait_For_Conversation (as on AIX or Linux systems) to wait for completion of the function. If you are writing a new CPI-C application for Windows systems, you should use this mechanism and not Wait_For_Conversation.

To use nonblocking calls and receive messages to indicate their completion, the application must issue the following calls before issuing a nonblocking call:

- `RegisterWindowMessage`, to obtain the message identifier that CPI-C will use for messages indicating completion of a nonblocking CPI-C function. This is a standard Windows function call, not specific to CPI-C; refer to your Windows documentation for more information about the function. The application must pass the value `WIN_CPIC_ASYNC_COMPLETE_MESSAGE` to the function; the returned value is a message identifier, as described below. (There is no need to issue the call again before subsequent CPI-C calls; the returned value will be the same for all calls issued by the application.)
- Set_Processing_Mode, to set the conversation's processing mode to `CM_NON_BLOCKING`.
- Specify_Windows_Handle, to specify the handle to which the completion message is sent.

Each time a nonblocking CPI-C function completes, CPI-C posts a message to the window handle specified on the Specify_Windows_Handle call. The format of the message is as follows:

- The message identifier is the value returned from the `RegisterWindowMessage` call.
- The *lParam* argument contains the conversation ID of the CPI-C call that has completed.
- The *wParam* argument contains the conversation *return_code* parameter from the CPI-C call that has completed. The possible values for this parameter depend on the individual call.

Function Call

```
void xchwnd (
             HWND                       hwnd,
             CM_RETURN_CODE CM_PTR      return_code
            );
```

Supplied Parameters

The supplied parameter is:

hwnd A window handle that CPI-C will use to post a message indicating that a nonblocking function has completed.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
The supplied parameter was not a valid Windows handle.

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

State When Issued

This call is not associated with a conversation.

State Change

There is no state change associated with this call.

When CPI-C sends a message to indicate that a nonblocking call has completed, the state change is dependent on the function that completed and its return code.

Test_Request_to_Send_Received (cmtrts)

The Test_Request_to_Send_Received call determines whether a request-to-send notification has been received from the partner program.

Function Call

```
void cmtrts (
             unsigned char CM_PTR                conversation_ID,
             CM_Request_to_Send_Received CM_PTR  request_to_send_received,
             CM_RETURN_CODE CM_PTR               return_code
            );
```

Function Call for Java CPI-C

AIX, LINUX

```
public native void cmtrts (
                           byte[]                        conversation_ID,
                           CPICControlInformationReceived request_to_send_received,
                           CPICReturnCode                return_code
                          );
```

Supplied Parameters

The supplied parameters are:

conversation_ID
This is the identifier for the conversation.

The value of this parameter is returned by the Initialize_Conversation, Initialize_For_Incoming, or Accept_Conversation call.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

request_to_send_received
This is the request-to-send-received indicator. Possible values are:

CM_REQ_TO_SEND_RECEIVED
The partner program has issued the Request_To_Send call, which requests the local program to change the conversation to Receive state.

CM_REQ_TO_SEND_NOT_RECEIVED
The partner program has not issued the Request_To_Send call.

This value is not relevant if the *return_code* parameter contains a value other than `CM_OK`.

return_code
Possible values are:

CM_OK The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
The value specified by *conversation_ID* is not valid.

CM_PROGRAM_STATE_CHECK
The conversation is in a state that is not valid.

For an explanation of the following return codes, see Appendix B, "Common Return Codes," on page 171.
`CM_OPERATION_NOT_ACCEPTED`
`CM_PRODUCT_SPECIFIC_ERROR`

State When Issued

The conversation must be in Receive, Send, Send-Pending, or Pending-Post state.

State Change

There is no state change.

Wait_For_Conversation (cmwait)

This function is not available in Java CPI-C. Java CPI-C functions always operate in blocking mode; that is, the function does not return control to the application until the requested processing has completed.

The Wait_For_Conversation call waits for completion of a previous CPI-C call that returned `CM_OPERATION_INCOMPLETE`.

If processing for the previous call has already finished when Wait_For_Conversation is issued, this call returns immediately; otherwise it blocks until CPI-C has finished processing the incomplete operation. If the application is involved in multiple conversations, this call waits on all conversations, and returns as soon as a call completes on any of them.

WINDOWS

New applications written for Windows systems should use Specify_Windows_Handle to obtain the results of nonblocking calls, instead of using Wait_For_Conversation. See "Specify_Windows_Handle (xchwnd)" on page 151. The Wait_For_Conversation call is provided for compatibility with other CPI-C implementations, but is not recommended for use by Windows applications.

In particular, if the application issues the Receive call in nonblocking mode and then issues other calls in nonblocking mode on the same conversation while Receive is outstanding, it must use Specify_Windows_Handle. It must not issue Wait_For_Conversation while more than one call is outstanding on the same conversation; the results of Wait_For_Conversation in this situation are undefined.

Function Call

```
void cmwait (
             unsigned char CM_PTR      conversation_ID,
             CM_INT32 CM_PTR           conversation_return_code,
             CM_RETURN_CODE CM_PTR     return_code
            );
```

Supplied Parameters

There are no supplied parameters for this call.

Returned Parameters

After the verb executes, CS/AIX returns parameters to indicate whether the execution was successful and, if not, to indicate the reason the execution was unsuccessful.

conversation_ID
: This is the identifier for the conversation on which the outstanding call completed.

conversation_return_code
: This is the return code from the completed call (which previously returned CM_OPERATION_INCOMPLETE). The possible values for this parameter depend on which call was outstanding. For more information, see the description of the specific call.

  This value is not relevant if the *return_code* parameter contains a value other than CM_OK.

return_code
: Possible values are:

  CM_OK The Wait_For_Conversation call executed successfully. The

conversation_return_code parameter indicates whether the previous incomplete operation completed successfully.

CM_PROGRAM_STATE_CHECK
There was no incomplete operation outstanding.

CM_PRODUCT_SPECIFIC_ERROR
For an explanation of this return code, see Appendix B, "Common Return Codes," on page 171.

WINDOWS

CM_SYSTEM_EVENT
The call was terminated by an operating system event, rather than by the completion of a previous CPI-C call.

State When Issued

The call is not associated with a specific conversation, so the conversation state is not relevant. However, the application must have at least one conversation with an incomplete operation outstanding.

State Change

If *return_code* is set to CM_OK, the state change depends on the outstanding call that completed, and on the return code from that call (the *conversation_return_code* parameter on this call). For more information, see the description of the specific call. If *return_code* is not CM_OK, there is no state change.

Usage Notes

This call does not change the program's current context (even if the outstanding operation that has completed is one that would normally do this, such as Accept_Incoming). If necessary, the program can use Extract_Conversation_Context for the *conversation_ID* returned on this call, to get the value of the conversation context, and Set_Conversation_Context to set its current context to this value.

If no previously outstanding call has completed, this call blocks (and the application's processing is suspended) until one completes.

AIX, LINUX

To check for completed calls without blocking, the application can use Check_For_Completion (which always returns immediately) to determine whether a call has completed, and call Wait_For_Conversation only when Check_For_Completion indicates that a call has completed (and therefore Wait_For_Conversation will return immediately).

If there are multiple outstanding calls (on different conversations), each Wait_For_Conversation call returns one outstanding call. After issuing Wait_For_Conversation, the application can check whether any other calls have completed by issuing Check_For_Completion.

WINDOWS

A Windows application can use Wait_For_Conversation, as described previously. However, the recommended method of handling nonblocking calls is to use Specify_Windows_Handle. This function, which must be issued before any nonblocking calls, specifies a Windows handle to which CPI-C sends a message when the call processing has completed. The application checks the results of the call when it receives this message, and does not use Wait_For_Conversation. Check_For_Completion, described previously for AIX or Linux systems, is not supported on Windows systems.

WinCPICCleanup

WINDOWS

The application uses this function to unregister as a Windows CPI-C user, after it has finished issuing CPI-C calls.

Function Call

```
BOOL WINAPI WinCPICCleanup (void);
```

Supplied Parameters

There are no supplied parameters for this call.

Returned Values

The return value from the function is one of the following:

TRUE The application was unregistered successfully.

FALSE An error occurred during processing of the call, and the application was not unregistered. Check the log files for messages indicating the cause of the failure.

WinCPICIsBlocking

WINDOWS

The application uses this function to check whether there is a blocking CPI-C call outstanding (a call issued with the conversation's processing mode set to CM_BLOCKING). For more information about blocking calls, see "Windows Considerations" on page 41.

Function Call

```
BOOL WINAPI WinCPICIsBlocking (void);
```

Supplied Parameters

There are no supplied parameters for this function.

Returned Values

The return value from the function is one of the following:

TRUE There is a blocking CPI-C call outstanding. If necessary, the application can use Cancel_Conversation or Deallocate to cancel the call and end the conversation.

FALSE There is no blocking CPI-C call outstanding.

WinCPICSetBlockingHook

WINDOWS

The application uses this call to specify its own blocking function, which CPI-C will use instead of the default blocking function. For more information about how the blocking function operates, and on the functions it must perform, see "Blocking Calls" on page 42.

Function Call

```
FARPROC WINAPI WinCPICSetBlockingHook (FARPROC lpBlockFunc);
```

Supplied Parameters

The supplied parameter is:

lpBlockFunc
The procedure instance address of the application's blocking function. The application should use the MakeProcInstance call to obtain this address; refer to your Windows documentation for more information.

Returned Values

The return value is the procedure instance address of the previous blocking function. If the application is using more than one blocking function, and will need to restore the previous blocking function later, it should save this address; it can then issue `WinCPICSetBlockingHook` again using the saved value, to restore the previous blocking function. If it is using only one blocking function, or will not need to restore the previous value, it can ignore the return value from this call.

Usage

The new blocking function remains in effect until the application issues one of the following calls:

- `WinCPICSetBlockingHook` (with a different procedure instance address), to specify a new blocking function or to restore a previous one
- `WinCPICUnhookBlockingHook` (described below), to stop using the current blocking function and return to the default blocking function.

WinCPICStartup

WINDOWS

The application uses this function to register as a Windows CPI-C user, and to determine whether the CPI-C software supports the Windows CPI-C version that it requires.

Function Call

```
int WINAPI WinCPICStartup (
                            WORD             wVersionRequired;
                            LPWCPICDATA      lpData;
                           )
typedef struct
{
  WORD        wVersion;
  char        szDescription[128];
} WCPICDATA;
```

Supplied Parameters

The supplied parameter is:

wVersionRequired
The version of Windows CPI-C that the application requires. CS/AIX supports version 1.0.

The low-order byte of this parameter specifies the major version number, and the high-order byte specifies the minor version number. For example:

Version	wVersionRequired
1.0	0x0001
1.1	0x0101
2.0	0x0002

If the application can use more than one version, it should specify the highest version that it can use.

Returned Values

The return value from the function is one of the following:

0 (zero)
The application was registered successfully, and the Windows CPI-C software supports either the version number specified by the application or a lower version. The application should check the version number in the WCPICDATA structure (see the description that follows) to ensure that it is high enough.

WCPICVERNOTSUPPORTED
The version number specified by the application was lower than the lowest version supported by the Windows CPI-C software. The application was not registered.

WCPICSYSNOTREADY
The CS/AIX software has not been started, or the local node is not active. The application was not registered.

If the return value from `WinCPICStartup` is `0` (zero), the `WCPICDATA` structure contains information about the support provided by the Windows CPI-C software. If the return value is nonzero, the contents of this structure are undefined and the application should not check them. The parameters in this structure are as follows:

wVersion
The Windows CPI-C version number that the software supports, in the same format as the *wVersionRequired* parameter (see the previous explanation). CS/AIX supports version 1.0.

If the software supports the requested version number, this parameter is set to the same value as the *wVersionRequired* parameter; otherwise it is set to the highest version that the software supports, which will be lower than the version number supplied by the application. The application must check the returned value and take action as follows:

- If the returned version number is the same as the requested version number, the application can use this Windows CPI-C implementation.
- If the returned version number is lower than the requested version number, the application can use this Windows CPI-C implementation but must not attempt to use features that are not supported by the returned version number. If it cannot do this because it requires features not available in the lower version, it should fail its initialization and not attempt to issue any CPI-C calls.

szDescription
A text string describing the Windows CPI-C software.

WinCPICUnhookBlockingHook

WINDOWS

The application uses this call to remove its own blocking function, which it has previously specified using `WinCPICSetBlockingHook`, and revert to using CPI-C's default blocking function.

Function Call

```
BOOL WINAPI WinCPICUnhookBlockingHook (void);
```

Supplied Parameters

There are no supplied parameters for this function.

Returned Values

The return value is one of the following:

TRUE The blocking function was removed successfully; any further blocking calls will use the default blocking function.

FALSE The call did not complete successfully.

WinCPICSetEvent

WINDOWS

The application uses this function to associate an event handle with verb completion for the specified conversation.

Function Call

```
VOID WINAPI WinCPICSetEvent (
                             unsigned char CM_PTR  conversation_ID,
                             HANDLE CM_PTR         event_handle,
                             CM_INT32 CM_PTR       return_code
                            );
```

Supplied Parameters

The supplied parameters are:

conversation_ID
: This is the identifier for the conversation for which this event is used. This parameter is returned by the initial Accept_Conversation call.

event_handle
: This is the handle of the event that is to be cleared when an asynchronous verb on the conversation completes. This parameter can replace an already-defined event or remove an already-defined event (by having NULL as the parameter).

Returned Parameters

return_code
: Possible values are:

  CM_OK The `WinCPICSetEvent` function executed successfully.

  CM_PROGRAM_PARAMETER_CHECK
  : One or more of the parameters passed to this function are invalid.

  CM_OPERATION_NOT_ACCEPTED
  : This value indicates that a previous operation on this conversation is incomplete and the `WinCPICSetEvent` call was not accepted.

Usage Notes

When a verb is issued on a nonblocking conversation, it returns `CM_OPERATION_INCOMPLETE` if it is going to complete asynchronously. If an event has been registered with the conversation, then the application can call `WaitForSingleObject` or `WaitForMultipleObjects` to be notified of the completion of the verb. When the verb has completed, the application must call Wait_for_Conversation to determine the return code for the asynchronous verb.

It is the responsibility of the application to reset the event.

WinCPICExtractEvent

WINDOWS

The application uses this function to determine the event handle being used for a CPI-C conversation.

Function Call

```
VOID WINAPI WinCPICExtractEvent (
                                 unsigned char CM_PTR  conversation_ID,
                                 HANDLE CM_PTR         event_handle,
                                 CM_INT32 CM_PTR       return_code
                                );
```

Supplied Parameters

The supplied parameter for this function is:

conversation_ID
This is the identifier for the conversation for which this event is used. This parameter is returned by the initial Accept_Conversation call.

Returned Parameters

event_handle
This is the handle of the event being used by this conversation. If no handle has been registered, this parameter returns a NULL value.

return_code
Possible values are:

CM_OK The `WinCPICExtractEvent` function executed successfully.

CM_PROGRAM_PARAMETER_CHECK
One or more of the parameters passed to this function are invalid.

Usage Notes

When a verb is issued on a nonblocking conversation, it returns `CM_OPERATION_INCOMPLETE` if it is going to complete asynchronously. If an event has been registered with the conversation, then the application can call `WaitForSingleObject` or `WaitForMultipleObjects` to be notified of the completion of the verb. `WinCPICExtractEvent` enables a CPI-C application to determine this event handle. When the verb has completed, the application must call Wait_for_Conversation to determine the return code for the asynchronous verb.

The Cancel_Conversation function can be called to cancel an operation and conversation.

If no event has been registered, then the asynchronous verb completes by posting a message to the window that the application has registered with the CPI-C library.

Chapter 4. Sample CPI-C Transaction Programs

This chapter describes the CS/AIX sample CPI-C transaction programs, which illustrate the use of CPI-C calls in AIX or Linux applications. For information on using CPI-C calls in a Java application, see Chapter 5, "Sample Java CPI-C Transaction Program," on page 167.

The following information is provided:

- Processing overview of the two programs
- Pseudocode for each program
- Instructions for compiling, linking, and running the two programs

Processing Overview

The programs presented in this chapter enable you to browse through a file on another system. The user is presented with a single data block at a time, in hexadecimal and character format. After each block, a user can request the next block, request the previous block, or quit.

CSAMPLE1 (the invoking program) sends a file name to CSAMPLE2 (the invoked program). If CSAMPLE2 locates the file, it returns the first block to CSAMPLE1; otherwise, it deallocates the conversation and ends.

If CSAMPLE1 receives a block, it displays the block on the screen and waits for the user to enter F for forward, B for backward, or Q for quit. If the user selects forward or backward, CSAMPLE1 sends the request to CSAMPLE2 which in turn sends the appropriate block. This process continues until the user selects the quit option, at which time CSAMPLE1 deallocates the conversation and both programs end.

If the user asks for the next block and CSAMPLE2 has sent the last one, CSAMPLE2 wraps to the beginning of file. Similarly, CSAMPLE2 wraps to send the last block if the user requests the previous one and the first block is displayed.

Neither program attempts to recover from errors. A bad return code from CPI-C causes the program to terminate with an explanatory message.

Pseudocode

This section contains the pseudocode for the transaction programs, CSAMPLE1 and CSAMPLE2.

The sample programs are provided as **csample1.c** and **csample2.c**, in the directory **/usr/lib/sna/samples** (AIX) or **/opt/ibm/sna/samples** (Linux).

CSAMPLE1 (Invoking Program)

The pseudocode for CSAMPLE1 (the invoking program) is as follows:

```
initialize
allocate
send data (data = filename)
do while no error and prompt not Q
    receive
```

```
    if data block received
        display data block
    else if permission to send received
        get user prompt (F, B, or Q)
        if prompt = F or B       /* Not Q */
            send data (data = prompt)
        endif
    endif
end do
deallocate
```

CSAMPLE2 (Invoked TP)

The pseudocode for CSAMPLE2 (the invoked TP) is as follows:

```
initialize
do while conversing
    receive
    if data received
        if first time (data = filename)
            open file
            if file not found
                deallocate
                set conversing false
            endif
        else (data = prompt)
            read and store prompt
        endif
        if (conversing)
            read file block
            send data (file block)
        endif
    else if deallocate received
        set conversing false
    endif
 end while conversing
close file
```

Testing the TPs

After examining the source code for CSAMPLE1 and CSAMPLE2, you may want to test the programs.

Although CPI-C is normally used for communications between programs on separate computers, you may find it convenient to run both programs on the same computer for testing purposes.

To compile and link the programs for an AIX or Linux system, take the following steps.

1. Copy the two files **csample1.c** and **csample2.c** from the directory **/usr/lib/sna/samples** (AIX) or **/opt/ibm/sna/samples** (Linux) to a private directory.
2. To compile and link the programs for AIX, use the following commands:

```
cc -o csample1 -I /usr/include/sna -bimport:/usr/lib/sna/cpic_r.exp csample1.c
cc -o csample2 -I /usr/include/sna -bimport:/usr/lib/sna/cpic_r.exp csample2.c
```

To compile and link the programs for Linux, use the following commands:

```
gcc -o csample1 -I /opt/ibm/sna/include -L /opt/ibm/sna/lib -lcpic -lappc -lsna_r -lpLiS -lpthread csample1.c
gcc -o csample2 -I /opt/ibm/sna/include -L /opt/ibm/sna/lib -lcpic -lappc -lsna_r -lpLiS -lpthread csample2.c
```

To run the programs, perform the following steps. Note that some of these steps involve updating the CS/AIX configuration, which is usually performed by the System Administrator.

The programs can run on the same computer, or on separate computers. In the following steps, the "source computer" is the computer where the invoking program CSAMPLE1 runs, and the "target computer" is the computer where the invoked program CSAMPLE2 runs.

1. If you are running the programs on separate computers, configure the communications link to support CP-CP sessions between the source and target computers. See *Communications Server for AIX Administration Guide* for more information.
2. Configure a mode with mode name LOCMODE.
3. Configure a logical unit (LU) on the source computer for CSAMPLE1 (the invoking program). Specify `TPLU1` as both the LU name and LU alias. Leave the default values for the other parameters.
4. Configure a symbolic destination name on the source computer. Do the following:
   - For *Name*, specify `CPICTEST`
   - For *Local LU*, select *Local LU alias* and specify `TPLU1` as the LU alias.
   - For *Partner LU*, specify the fully-qualified name *netname*.**TPLU2**, where *netname* is the SNA network name of the target computer.
   - For *Mode*, specify `LOCMODE`.
   - For *Partner TP*, specify `TPNAME2`.

   Leave the default values for other parameters.
5. Configure an LU on the target computer for CSAMPLE2 (the invoked program). Specify `TPLU2` as both the LU name and LU alias. Leave the default values for the other parameters.
6. Configure the invoked TP in the CS/AIX invokable TP data file on the target computer. Refer to the *Communications Server for AIX Administration Guide* for more information.
   - For the *TP name* parameter, specify `TPNAME2` (the name specified by the invoking TP).
   - For *Full path to TP executable*, enter the full path name of the executable file **csample2**.
   - For the *User ID* parameter, specify your AIX user ID on the target computer.
   - Leave the default values for other parameters.
7. If the invoked TP is to run with a *user_id* of `root`, change the permissions on the executable file to allow it to do so. Use the following command:

   chmod +s csample2
8. Start the CS/AIX software using this configuration file.
9. Set the following environment variables:
   - `APPCLLU` to `TPLU1` (the name of the local LU for **csample1**)
   - `APPCTPN` to `TPNAME1`
10. Start the invoking program, **csample1**. This program requires one parameter, the full path name (on the target computer) of the file to be displayed. For example:

```
csample1 /usr/jim/myfile
```

11. Enter `F` or `B` to display blocks of the requested file. Use `Q` to end the invoking program; the invoked program will end as well.

Chapter 5. Sample Java CPI-C Transaction Program

AIX, LINUX

This chapter describes the CS/AIX sample Java CPI-C transaction program **JPing**, which illustrates the use of CPI-C calls in a Java application. For information on using CPI-C calls in a standard C program, see Chapter 4, "Sample CPI-C Transaction Programs," on page 163.

The following information is provided:

- Overview of the program
- Instructions for compiling, linking, and running the program

Overview

The sample Java CPI-C program, **JPing** (in the file **/usr/lib/sna/samples/JPing.java** (AIX) or **/opt/ibm/sna/samples/JPing.java** (Linux)) is a simple Java implementation of the standard APPC function **aping**, which is used to check connectivity with a remote node. For more information about **aping**, refer to the *Communications Server for AIX APPC Application Suite User's Guide* or the *Communications Server for AIX Administration Command Reference*.

You can optionally specify a symbolic destination name identifying the partner LU to be contacted, the number of ping iterations to be attempted, and the size of the information sent at each iteration.

For more information about the operation of the program, see the comments in the program source file.

Compiling, Linking, and Running the Sample Program

After examining the source code for **JPing**, you may want to build and test the program.

Before compiling and linking the application, specify the directory where Java classes are stored. To do this, set and export the environment variable `CLASSPATH` to **/usr/lib/sna/java/cpic.jar:.** (AIX) or **/opt/ibm/sna/java/cpic.jar:.** (Linux).

To compile and link the program, take the following steps.

1. Copy the file **JPing.java** from the directory **/usr/lib/sna/samples** (AIX) or **/opt/ibm/sna/samples** (Linux)to a private directory.
2. From the private directory, compile and link the application using the Java compiler **javac** in the normal way, using the following command:

   javac JPing.java

You should see that the file **JPing.class** has been generated.

Before running a Java CPI-C application, you need to specify the directory where libraries are stored, so that the application can find them at run time.

To do this, set and export the appropriate environment variable as follows.

export LD_LIBRARY_PATH=/usr/lib/sna

You may also need to set and export the APPCTPN environment variable to specify the local TP name for the application, as described in "Specifying the Local TP Name" on page 33.

Running the program involves updating the CS/AIX configuration to include a symbolic destination name identifying the partner LU. This task is usually performed by the System Administrator. The following steps are required:

- For Symbolic Destination Name, specify JPING.
- For Partner TP Name Type, specify Application Program.
- For Partner TP Name, specify APINGD.
- For Partner LU, specify the fully-qualified name of the partner LU you want to contact.
- For Mode Name, specify #INTER.

Leave the default values for other parameters.

Run the application using the Java interpreter **java** in the normal way. Use the following command:

java JPing [*sym_dest_name*] [
-i *num_iterations*] [**-s** *data_len*]

sym_dest_name indicates the symbolic destination name to be used by the program. If you do not specify this option, the default is JPING.

The **-i** option indicates the number of ping iterations to be performed. If you do not specify this option, the default is 2.

The **-s** option indicates the number of bytes of data to be sent to the partner program. If you do not specify this option, the default is 100.

For more information about how the number of ping iterations and the data length are used, refer to the description of **aping** in the *Communications Server for AIX APPC Application Suite User's Guide* or the *Communications Server for AIX Administration Command Reference*.

Appendix A. Return Code Values

This appendix lists all the possible return codes in the CPI-C interface in numerical order. The values are defined in the header file **cmc.h**(for AIX or Linux) or **wincpic.h** (for Windows).

You can use this appendix as a reference to check the meaning of a return code received by your application.

```
CM_OK                                0
CM_ALLOCATE_FAILURE_NO_RETRY         1
CM_ALLOCATE_FAILURE_RETRY            2
CM_CONVERSATION_TYPE_MISMATCH        3
CM_PIP_NOT_SPECIFIED_CORRECTLY       5
CM_SECURITY_NOT_VALID                6
CM_SYNC_LVL_NOT_SUPPORTED_LU         7
CM_SYNC_LVL_NOT_SUPPORTED_PGM        8
CM_TPN_NOT_RECOGNIZED                9
CM_TP_NOT_AVAILABLE_NO_RETRY        10
CM_TP_NOT_AVAILABLE_RETRY           11
CM_DEALLOCATED_ABEND                17
CM_DEALLOCATED_NORMAL               18
CM_PARAMETER_ERROR                  19
CM_PRODUCT_SPECIFIC_ERROR           20
CM_PROGRAM_ERROR_NO_TRUNC           21
CM_PROGRAM_ERROR_PURGING            22
CM_PROGRAM_ERROR_TRUNC              23
CM_PROGRAM_PARAMETER_CHECK          24
CM_PROGRAM_STATE_CHECK              25
CM_RESOURCE_FAILURE_NO_RETRY        26
CM_RESOURCE_FAILURE_RETRY           27
CM_UNSUCCESSFUL                     28
CM_DEALLOCATED_ABEND_SVC            30
CM_DEALLOCATED_ABEND_TIMER          31
CM_SVC_ERROR_NO_TRUNC               32
CM_SVC_ERROR_PURGING                33
CM_SVC_ERROR_TRUNC                  34
CM_OPERATION_INCOMPLETE             35
CM_SYSTEM_EVENT                     36
CM_OPERATION_NOT_ACCEPTED           37
CM_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
CM_SECURITY_MUTUAL_FAILED           47
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
```

Appendix B. Common Return Codes

This appendix describes the return codes that are common to several CPI-C calls. The return codes are listed in alphabetical order. Return codes generated when the partner program is a non-CPI-C LU 6.2 program are listed separately.

Call-specific return codes are described in the documentation for the individual calls in Chapter 3, "CPI-C Calls," on page 47.

Return Codes from Any Partner Program

The following return codes can occur with any partner program. (Other return codes, which can only occur when the partner program is not a CPI-C program, are listed separately.)

CM_ALLOCATION_FAILURE_NO_RETRY
The conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the System Administrator should examine the error log file. Do not attempt to retry the allocation until the error has been corrected.

CM_ALLOCATION_FAILURE_RETRY
The conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

AIX, LINUX

CM_CALL_NOT_SUPPORTED
This return code is used only in Java CPI-C applications.

The application used a CPI-C function that is defined in the Java CPI-C class but is not supported.

CM_CONVERSATION_TYPE_MISMATCH
The partner LU or program does not support the conversation type (basic or mapped) specified in the allocation request.

CM_DEALLOCATED_ABEND
The conversation has been deallocated for one of the following reasons:

- The partner program has issued the Deallocate call with the deallocate type set to CM_DEALLOCATE_ABEND. If the conversation is in Receive state for the partner program when this call is issued by the local program, data sent by the local program and not yet received by the partner program is purged.
- The partner program has terminated normally but did not deallocate the conversation before terminating.
- The local program issued the Cancel_Conversation call, which cancels all outstanding asynchronous CPI-C calls on the conversation.

CM_DEALLOCATED_NORMAL
This return code does not indicate an error.

The partner program issued the Deallocate call with the deallocate type set to one of the following:

- CM_DEALLOCATE_FLUSH
- CM_DEALLOCATE_SYNC_LEVEL with the synchronization level of the conversation specified as CM_NONE

CM_OK The call executed successfully.

CM_OPERATION_INCOMPLETE

The call was issued successfully, and is operating in nonblocking mode; that is, control has been returned to the program even though processing for the call has not yet completed.

The program can continue with any processing not related to this conversation (including issuing CPI-C calls on other conversations). On this conversation, it can issue a limited range of CPI-C calls (such as the Extract_* calls). This is different from the IBM CPI-C 2.0 specification in which the program cannot issue any calls on this conversation except Wait_For_Conversation or Cancel_Conversation.

AIX, LINUX

At a later time, the application can issue Check_For_Completion to determine whether the outstanding nonblocking call has completed, Wait_For_Conversation to wait for it to complete, or Cancel_Conversation to cancel the outstanding call and end the conversation.

WINDOWS

If the application has used Specify_Windows_Handle to receive notification of asynchronous call completion, it should not issue further calls on this conversation until it has received this notification. Otherwise, the application can issue Wait_For_Conversation to wait for the nonblocking call to complete, or Cancel_Conversation to cancel the outstanding call and end the conversation.

CM_OPERATION_NOT_ACCEPTED

The call cannot be issued because of one of the following conditions:

- There is a nonblocking call outstanding on this conversation. The program can continue with any processing not related to this conversation (including issuing CPI-C calls on other conversations), but cannot issue most CPI-C calls on this conversation.

  AIX, LINUX

  At a later time, the application can issue Check_For_Completion to determine whether an outstanding nonblocking call has completed, Wait_For_Conversation to wait for it to complete, or Cancel_Conversation to cancel the outstanding call and end the conversation.

- The program is running in a DCE multi-threaded environment, and there is a call outstanding on this conversation from another thread of the program. Only one call for each conversation can be outstanding at any one time.

  WINDOWS

  If the application has used Specify_Windows_Handle to receive notification of asynchronous call completion, it should not issue further calls on this conversation until it has received this notification. Otherwise, the application can issue Wait_For_Conversation to wait for the nonblocking call to complete, or Cancel_Conversation to cancel the outstanding call and end the conversation.

CM_PARAMETER_ERROR
A parameter referred to by CPI-C is not valid. The parameter that is not valid is one that can be supplied either by the program or by another component outside the program's control (such as the configuration file). For example, the *mode_name* parameter may have been specified by the program using Set_Mode_Name, or may have been taken from the side information entry specified by the *sym_dest_name* parameter.

CM_PRODUCT_SPECIFIC_ERROR
When CPI-C generates a CM_PRODUCT_SPECIFIC_ERROR return code, it makes an entry in the log file indicating the cause of the error and any action required. Refer to the *Communications Server for AIX Administration Guide* for more information about interpreting these messages.

CM_PROGRAM_ERROR_NO_TRUNC
The partner program has issued the Send_Error call while in Send state or in Send-Pending state with the error direction set to CM_SEND_ERROR. Data was not truncated.

CM_PROGRAM_ERROR_PURGING
One of the following conditions has occurred:

- The partner program issued the Send_Error call while in Receive or Confirm state. Data sent but not yet received is purged.
- The partner program has issued the Send_Error call while in Send-Pending state with the error direction set to CM_RECEIVE_ERROR. Data was not purged.

CM_PROGRAM_ERROR_TRUNC
The partner program in a basic conversation has issued a Send_Error call while in Send state, before finishing sending a complete logical record. The local program may have received the first part of the logical record through a Receive call.

CM_PROGRAM_PARAMETER_CHECK
The program supplied a parameter that is not valid to the call. For details, see individual calls in Chapter 3, "CPI-C Calls," on page 47.

CM_PROGRAM_STATE_CHECK
The call issued is not allowed in the current conversation state, or is not appropriate because of the current setting of a conversation characteristic. For details, see individual calls in Chapter 3, "CPI-C Calls," on page 47.

CM_RESOURCE_FAILURE_NO_RETRY
One of the following conditions has occurred:

- The conversation was terminated prematurely because of a permanent condition. Do not attempt to retry until the error has been corrected.
- The partner program did not deallocate the conversation before terminating normally.

CM_RESOURCE_FAILURE_RETRY
The conversation was terminated prematurely because of a temporary condition, such as modem failure. Retry the conversation.

CM_SECURITY_NOT_VALID
The user ID or password specified in the allocation request was not accepted by the partner LU.

CM_SYNC_LVL_NOT_SUPPORTED_PGM
The partner program does not support the synchronization level specified in the allocation request.

CM_SYNC_LVL_NOT_SUPPORTED_LU
The partner LU does not support the synchronization level specified in the allocation request.

CM_TP_NOT_AVAILABLE_NO_RETRY
The partner LU cannot start the program specified in the allocation request because of a permanent condition. The reason for the error may be logged on the remote node. Do not retry the allocation until the cause of the error has been corrected.

CM_TP_NOT_AVAILABLE_RETRY
The partner LU cannot start the program specified in the allocation request because of a temporary condition. The reason for the error may be logged on the remote node. Retry the allocation.

CM_TPN_NOT_RECOGNIZED
The partner LU does not recognize the program name specified in the allocation request.

CM_UNSUCCESSFUL
The call was not executed successfully. This return code occurs in the following cases:

- The program issued Allocate with the *return_control* parameter set to `CM_IMMEDIATE`, and CS/AIX was unable to assign a session for the conversation immediately.
- The program issued Receive with the *receive_type* parameter set to `CM_RECEIVE_IMMEDIATE`, and no data or control information from the partner program was currently available.

  AIX, LINUX

- The program issued Check_For_Completion, and no outstanding nonblocking function had completed on any of the program's conversations.

Non-CPI-C LU 6.2 Partner Program

The following return codes can occur when the partner program is a non-CPI-C LU 6.2 program, for example an APPC TP. The verbs described in these paragraphs are LU 6.2 verbs.

CM_DEALLOCATED_ABEND_SVC
The conversation has been deallocated for one of the following reasons:

- The partner program has issued the DEALLOCATE verb with *TYPE* set to ABEND_SVC.
- The partner program did not deallocate the conversation before terminating.

If the conversation is in Receive state for the partner program when this call is issued by the local program, data sent by the local program and not yet received by the partner program is purged.

CM_DEALLOCATED_ABEND_TIMER
The conversation has been deallocated because the partner program has issued the DEALLOCATE verb with *TYPE* set to ABEND_TIMER. If the conversation is in Receive state for the partner program when this call is issued by the local program, data sent by the local program and not yet received by the partner program is purged.

CM_PIP_NOT_SPECIFIED_CORRECTLY
The allocation request was rejected by a non-CPI-C LU 6.2 program. The partner program requires one or more PIP data variables, and CPI-C does not support PIP data.

CM_SVC_ERROR_NO_TRUNC
The partner program (or partner LU) issued a SEND_ERROR verb with the *TYPE* parameter set to SVC during a basic conversation while in Send state. Data was not truncated.

CM_SVC_ERROR_PURGING
While in Send state, the partner program (or partner LU) issued a SEND_ERROR verb with the *TYPE* parameter set to SVC. Data sent to the partner program may have been purged.

CM_SVC_ERROR_TRUNC
The partner program (or partner LU) in a basic conversation issued a SEND_ERROR verb with the *TYPE* parameter set to SVC while in Recieve or Confirm state, before finishing sending a complete logical record. The local program may have received the first part of the logical record.

Appendix C. Conversation State Changes

Table 25 on page 178 shows the conversation states in which each CPI-C function call can be issued, and the state change which occurs on completion of the call.

In some cases, the state change depends on the return code from the call; in most cases, there is no state change for non-OK return codes. Where no return codes are shown, a return code of CM_OK causes the state change shown, and any non-OK return code causes no state change (except as described in the note that follows). Where there are different state changes according to the return code, the applicable values are listed in the Return codes column.

The possible conversation states are shown as column headings. Against each call, the following information is given under each heading to indicate the results of issuing the call in this state:

X The call cannot be issued in this state.

T, I, II, S, SP, R, C, CS, CD, or PP
Indicates the state of the conversation after the call has completed: Reset (R), Initialize (I), Initialize-Incoming (II), Send (S), Send-Pending (SP), Receive (R), Confirm (C), Confirm-Send (CS), Confirm-Deallocate (CD), or Pending-Post (PP).

(blank)
The return code shown cannot occur in this state.

See function
See the description of this function in Chapter 3, "CPI-C Calls," on page 47. The changes in the conversation state depend on the returned parameters from the call.

Note: The conversation will always enter Reset state if any of the following return codes are received:

- CM_ALLOCATION_FAILURE_NO_RETRY, CM_ALLOCATION_FAILURE_RETRY
- CM_CONVERSATION_TYPE_MISMATCH
- CM_DEALLOCATED_NORMAL, CM_DEALLOCATED_ABEND
- CM_PIP_NOT_SPECIFIED_CORRECTLY
- CM_RESOURCE_FAILURE_RETRY, CM_RESOURCE_FAILURE_NO_RETRY
- CM_SECURITY_NOT_VALID, CM_SYNC_LVL_NOT_SUPPORTED_PGM, CM_SYNC_LVL_NOT_SUPPORTED_LU
- CM_TPN_NOT_RECOGNIZED, CM_TP_NOT_AVAILABLE_RETRY, CM_TP_NOT_AVAILABLE_NO_RETRY

AIX, LINUX

Pending-Post state does not apply to AIX or Linux systems. All references to this state should be ignored.

WINDOWS

Initialize-Incoming state does not apply to Windows systems. All references to this state should be ignored.

The Windows-specific function calls are not associated with a particular conversation, and have no effect on conversation states. They are not listed in this appendix.

Table 25. Conversation State Changes

CPI-C Call and *primary_rc* Values	**State in Which Issued**									
	Reset (T)	**Init (I)**	**Init-Inc (II)**	**Send (S)**	**Send Pend (SP)**	**Recv (R)**	**Confm (C)**	**Confm Send (CS)**	**Confm Deall (CD)**	**Pend Post (PP)**
Accept_Conversation	R	X	X	X	X	X	X	X	X	X
Accept_Incoming	X	X	R	X	X	X	X	X	X	X
Allocate	X		X	X	X	X	X	X	X	X
CM_OK		S								
(Allocate failure)		T								
Cancel_Conversation	X	T	T	T	T	T	T	T	T	T
Check_For_Completion	T	I	II	S	SP	R	C	CS	CD	X
Confirm	X	X	X			X	X	X	X	X
CM_OK				S	S					
(Program error, SVC error)				R	R					
Confirmed	X	X	X	X	X	X	R	S	T	X
Convert_Incoming, Convert_Outgoing	T	I	II	S	SP	R	C	CS	CD	X
Deallocate (Abend)	X									
CM_OK		T	T	T	T	T	T	T	T	T
(Program error, SVC error)		R	R	R	R	R	R	R	R	R
Deallocate (other)	X	X	X			X	X	X	X	X
CM_OK				T	T					
(Program error, SVC error)				R	R					
Delete_CPIC_ Side_Information	T	I	II	S	SP	R	C	CS	CD	X
Extract_Conversation_ Context	X	X	X	S	SP	R	C	CS	CD	X
Extract_Conversation_ Security_Type	X	I	II	S	SP	R	C	CS	CD	X
Extract_Conversation_ State	X	I	II	S	SP	R	C	CS	CD	X

Table 25. Conversation State Changes (continued)

CPI-C Call and *primary_rc* Values	**State in Which Issued**									
	Reset (T)	**Init (I)**	**Init-Inc (II)**	**Send (S)**	**Send Pend (SP)**	**Recv (R)**	**Confm (C)**	**Confm Send (CS)**	**Confm Deall (CD)**	**Pend Post (PP)**
Extract_Conversation_Type	X	I	II	S	SP	R	C	CS	CD	X
Extract_CPIC_Side_Information	T	I	II	S	SP	R	C	CS	CD	X
Extract_Local_LU_Name	X	I	II	S	SP	R	C	CS	CD	X
Extract_Maximum_Buffer_Size	T	I	II	S	SP	R	C	CS	CD	X
Extract_Mode_Name	X	I	II	S	SP	R	C	CS	CD	X
Extract_Partner_LU_Name	X	I	X	S	SP	R	C	CS	CD	X
Extract_Security_User_ID	X	I	II	S	SP	R	C	CS	CD	X
Extract_Sync_Level	X	I	X	S	SP	R	C	CS	CD	X
Extract_TP_Name	X	I	X	S	SP	R	C	CS	CD	X
Flush	X	X	X	S	S	X	X	X	X	X
Initialize_Conversation	I	X	X	X	X	X	X	X	X	X
Initialize_For_Incoming	II	X	X	X	X	X	X	X	X	X
Prepare_To_Receive	X	X	X			X	X	X	X	X
`CM_OK`, Program error, SVC error				R	R					
Receive (receive type `CM_RECEIVE_IMMEDIATE`)	X	X	X	X	X	See function	X	X	X	X
Receive (receive type `CM_RECEIVE_AND_WAIT`)	X	X	X	See function	See function	See function	X	X	X	X
Release_Local_TP_Name	I	X	X	X	X	X	X	X	X	X
Request_To_Send	X	X	X	S	SP	R	C	CS	CD	PP
Send_Data	X	X	X	See function	See function	X	X	X	X	X
Send_Error	X	X	X							
`CM_OK`				S	S	S	S	S	S	S
Program error, SVC error				R	R	R	R	R	R	PP
Set_Conversation_Context	X	I	II	S	SP	R	C	CS	CD	X

Table 25. Conversation State Changes (continued)

CPI-C Call and *primary_rc* Values	**State in Which Issued**									
	Reset (T)	**Init (I)**	**Init-Inc (II)**	**Send (S)**	**Send Pend (SP)**	**Recv (R)**	**Confm (C)**	**Confm Send (CS)**	**Confm Deall (CD)**	**Pend Post (PP)**
Set_Conversation_Security_Password	X	I	X	X	X	X	X	X	X	X
Set_Conversation_Security_Type	X	I	X	X	X	X	X	X	X	X
Set_Conversation_Security_User_ID	X	I	X	X	X	X	X	X	X	X
Set_Conversation_Type	X	I	X	X	X	X	X	X	X	X
Set_CPIC_Side_Information	T	I	II	S	SP	R	C	CS	CD	X
Set_Deallocate_Type	X	I	II	S	SP	R	C	CS	CD	X
Set_Error_Direction	X	I	II	S	SP	R	C	CS	CD	X
Set_Fill	X	I	II	S	SP	R	C	CS	CD	X
Set_Local_LU_Name	X	I	X	X	X	X	X	X	X	X
Set_Log_Data	X	I	II	S	SP	R	C	CS	CD	X
Set_Mode_Name	X	I	X	X	X	X	X	X	X	X
Set_Partner_LU_Name	X	I	X	X	X	X	X	X	X	X
Set_Prepare_To_Receive_Type	X	I	II	S	SP	R	C	CS	CD	X
Set_Processing_Mode	X	I	II	S	SP	R	C	CS	CD	X
Set_Receive_Type	X	I	II	S	SP	R	C	CS	CD	X
Set_Return_Control	X	I	X	X	X	X	X	X	X	X
Set_Send_Type	X	I	II	S	SP	R	C	CS	CD	X
Set_Sync_Level	X	I	X	X	X	X	X	X	X	X
Set_TP_Name	X	I	X	X	X	X	X	X	X	X
Specify_Local_TP_Name	T	I	II	S	SP	R	C	CS	CD	X
Test_Request_To_Send_Received	X	X	X	S	SP	R	X	X	X	PP
Wait_For_Conversation	Can be issued in any state; new state depends on the outstanding call that completed, and the return code from that call. See the information for the appropriate call.									

Appendix D. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation, Site Counsel
P.O. Box 12195
3039 Cornwallis Road
Research Triangle Park, NC 27709-2195
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

ACF/VTAM
Advanced Peer-to-Peer Networking
AIX
AIXwindows
AnyNet
Application System/400
APPN
AS/400
CICS
DATABASE 2
DB2
Enterprise System/3090
Enterprise System/4381
Enterprise System/9000
ES/3090
ES/9000
eServer
IBM
IBMLink
IMS
MVS
MVS/ESA
Operating System/2
Operating System/400
OS/2
OS/400
PowerPC
PowerPC Architecture
pSeries
S/390
System/390
VSE/ESA
VTAM
WebSphere
zSeries

The following terms are trademarks or registered trademarks of other companies:

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc., in the United States, other countries, or both.

UNIX is a registered trademark in the United States and other countries licensed exclusively through The Open Group.

Intel is a trademark of Intel Corporation.

Linux is a trademark of Linus Torvalds.

RedHat and RPM are trademarks of Red Hat, Inc.

SuSE Linux is a trademark of SuSE Linux AG.

UnitedLinux is a trademark of UnitedLinux LLC.

Microsoft, Windows, Windows NT, Windows 2003, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Bibliography

The following IBM publications provide information about the topics discussed in this library. The publications are divided into the following broad topic areas:

- CS/AIX, Version 6.3
- IBM Communications Server for AIX, Version 4 Release 2
- Redbooks™
- AnyNet/2 and SNA
- Block Multiplexer and S/390 ESCON Channel PCI Adapter
- AIX operating system
- Systems Network Architecture (SNA)
- Host configuration
- z/OS® Communications Server
- Multiprotocol Transport Networking
- Transmission Control Protocol/Internet Protocol (TCP/IP)
- X.25
- Advanced Program-to-Program Communication (APPC)
- Programming
- Other IBM networking topics

For books in the CS/AIX library, brief descriptions are provided. For other books, only the titles, order numbers, and, in some cases, the abbreviated title used in the text of this book are shown here.

CS/AIX Version 6.3Publications

The CS/AIX library comprises the following books. In addition, softcopy versions of these documents are provided on the CD-ROM. See *IBM Communications Server for AIX Quick Beginnings* for information about accessing the softcopy files on the CD-ROM. To install these softcopy books on your system, you require 9–15 MB of hard disk space (depending on which national language versions you install).

- *IBM Communications Server for AIX Migration Guide* (SC31-8585)

  This book explains how to migrate from Communications Server for AIX Version 4 Release 2 or earlier to CS/AIX Version 6.
- *IBM Communications Server for AIX Quick Beginnings* (GC31-8583)

  This book is a general introduction to CS/AIX, including information about supported network characteristics, installation, configuration, and operation.
- *IBM Communications Server for AIX Administration Guide* (SC31-8586)

  This book provides an SNA and CS/AIX overview and information about CS/AIX configuration and operation.
- *IBM Communications Server for AIX Administration Command Reference* (SC31-8587)

  This book provides information about SNA and CS/AIX commands.
- *IBM Communications Server for AIX CPI-C Programmer's Guide* (SC31-8591)

  This book provides information for experienced "C" or Java programmers about writing SNA transaction programs using the CS/AIX CPI Communications API.
- *IBM Communications Server for AIX APPC Programmer's Guide* (SC31-8590)

This book contains the information you need to write application programs using Advanced Program-to-Program Communication (APPC).

- *IBM Communications Server for AIX LUA Programmer's Guide* (SC31-8592)

  This book contains the information you need to write applications using the Conventional LU Application Programming Interface (LUA).
- *IBM Communications Server for AIX CSV Programmer's Guide* (SC31-8593)

  This book contains the information you need to write application programs using the Common Service Verbs (CSV) application program interface (API).
- *IBM Communications Server for AIX MS Programmer's Guide* (SC31-8594)

  This book contains the information you need to write applications using the Management Services (MS) API.
- *IBM Communications Server for AIX NOF Programmer's Guide* (SC31-8595)

  This book contains the information you need to write applications using the Node Operator Facility (NOF) API.
- *IBM Communications Server for AIX Diagnostics Guide* (SC31-8588)

  This book provides information about SNA network problem resolution.
- *IBM Communications Server for AIX AnyNet® Guide to APPC over TCP/IP* (GC31-8598)

  This book provides installation, configuration, and usage information for the AnyNet APPC over TCP/IP function of CS/AIX.
- *IBM Communications Server for AIX AnyNet Guide to Sockets over SNA* (GC31-8597)

  This book provides installation, configuration, and usage information for the AnyNet Sockets over SNA function of CS/AIX.
- *IBM Communications Server for AIX APPC Application Suite User's Guide* (SC31-8596)

  This book provides information about APPC applications used with CS/AIX.
- *IBM Communications Server for AIX Glossary* (GC31-8589)

  This book provides a comprehensive list of terms and definitions used throughout the IBM Communications Server for AIX library.

IBM Communications Server for AIX Version 4 Release 2 Publications

The following book is from a previous release of Communications Server for AIX, and does not apply to Version 6. You may find this book useful as a reference for information that is still supported, but not included in Version 6.

- *IBM Communications Server for AIX Transaction Program Reference.* (SC31-8212)

  This book provides Version 4 Release 2 information about the transaction programming APIs. Applications written to use the Version 4 Release 2 APIs can still be used with Version 6.

IBM Redbooks

IBM maintains an International Technical Support Center that produces publications known as Redbooks. Similar to product documentation, Redbooks cover theoretical and practical aspects of SNA technology. However, they do not include the information that is supplied with purchased networking products.

The following books contain information that may be useful for CS/AIX:

- *IBM Communications Server for AIX Version 6* (SG24-5947)

- *IBM CS/AIX Understanding and Migrating to Version 5: Part 2 - Performance* (SG24-2136)
- *Load Balancing for Communications Servers* (SG24-5305)

On the World Wide Web, users can download Redbook publications by using http://www.redbooks.ibm.com.

Block Multiplexer and S/390 ESCON Channel PCI Adapter publications

The following books contain information about the Block Multiplexer and the S/390 ESCON Channel PCI Adapter:

- AIX Version 4.1 Block Multiplexer Channel Adapter: User's Guide and Service Information (SC31-8196)
- AIX Version 4.1 Enterprise Systems Connection Adapter: User's Guide and Service Information (SC31-8196)
- AIX Version 4.3 S/390 ESCON Channel PCI: User's Guide and Service Information (SC23-4232)
- IBM Communications Server for AIX Channel Connectivity User's Guide (SC31-8219)

AnyNet/2 Sockets and SNA publications

The following books contain information about AnyNet/2 Sockets and SNA

- AnyNet/2 Version 2.0: Guide to Sockets over SNA (GV40-0376)
- AnyNet/2 Version 2.0: Guide to SNA over TCP/IP (GV40-0375)
- AnyNet/2: Guide to Sockets over SNA Gateway Version 1.1 (GV40-0374)
- z/OS V1R2.0 Communications Server: AnyNet Sockets over SNA (SC31-8831)
- z/OS V1R2.0 Communications Server: AnyNet SNA over TCP/IP (SC31-8832)

AIX Operating System Publications

The following books contain information about the AIX operating system:

- *AIX Version 5.3 System Management Guide: Operating System and Devices* (SC23-4910)
- *AIX Version 5.3 System Management Concepts: Operating System and Devices* (SC23-4908)
- *AIX Version 5.3 System Management Guide: Communications and Networks* (SC23-4909)
- *AIX Version 5.3 Performance Management Guide* (SC23-4905)
- *AIX Version 5.3 Performance Tools Guide and Reference* (SC23-4906)
- *Performance Toolbox Version 2 and 3 Guide and Reference* (SC23-2625)
- *AIXlink/X.25 Version 2.1 for AIX: Guide and Reference* (SC23-2520)

Systems Network Architecture (SNA) Publications

The following books contain information about SNA networks:

- *Systems Network Architecture: Format and Protocol Reference Manual—Architecture Logic for LU Type 6.2* (SC30-3269)
- *Systems Network Architecture: Formats* (GA27-3136)
- *Systems Network Architecture: Guide to SNA Publications* (GC30-3438)

- *Systems Network Architecture: Network Product Formats* (LY43-0081)
- *Systems Network Architecture: Technical Overview* (GC30-3073)
- *Systems Network Architecture: APPN Architecture Reference* (SC30-3422)
- *Systems Network Architecture: Sessions between Logical Units* (GC20-1868)
- *Systems Network Architecture: LU 6.2 Reference—Peer Protocols* (SC31-6808)
- *Systems Network Architecture: Transaction Programmer's Reference Manual for LU Type 6.2* (GC30-3084)
- *Systems Network Architecture: 3270 Datastream Programmer's Reference* (GA23-0059)
- *Networking Blueprint Executive Overview* (GC31-7057)
- *Systems Network Architecture: Management Services Reference* (SC30-3346)

Host Configuration Publications

The following books contain information about host configuration:

- *ES/9000, ES/3090 IOCP User's Guide Volume A04* (GC38-0097)
- *3174 Establishment Controller Installation Guide* (GG24-3061)
- *3270 Information Display System 3174 Establishment Controller: Planning Guide* (GA27-3918)
- *OS/390 Hardware Configuration Definition (HCD) User's Guide* (SC28-1848)
- *ESCON Director Planning* (GA23-0364)

z/OS Communications Server Publications

The following books contain information about z/OS Communications Server:

- *z/OS V1R7 Communications Server: SNA Network Implementation Guide* (SC31-8777-05)
- *z/OS V1R7 Communications Server: SNA Diagnostics* (Vol 1: GC31-6850-00, Vol 2: GC31-6851-00)
- *z/OS V1R6 Communications Server: Resource Definition Reference* (SC31-8778-04)

Multiprotocol Transport Networking publications

The following books contain information about Multiprotocol Transport Networking architecture:

- Multiprotocol Transport Networking: Formats (GC31-7074)
- Multiprotocol Transport Networking Architecture: Technical Overview (GC31-7073)

TCP/IP Publications

The following books contain information about the Transmission Control Protocol/Internet Protocol (TCP/IP) network protocol:

- *z/OS V1R7 Communications Server: IP Configuration Guide* (SC31-8775-07)
- *z/OS V1R7 Communications Server: IP Configuration Reference* (SC31-8776-08)
- *z/VM V5R1 TCP/IP Planning and Customization* (SC24-6125-00)

X.25 Publications

The following books contain information about the X.25 network protocol:

- *AIXLink/X.25 for AIX: Guide and Reference* (SC23-2520)
- *RS/6000® AIXLink/X.25 Cookbook* (SG24-4475)
- *Communications Server for OS/2 Version 4 X.25 Programming* (SC31-8150)

APPC Publications

The following books contain information about Advanced Program-to-Program Communication (APPC):

- *APPC Application Suite V1 User's Guide* (SC31-6532)
- *APPC Application Suite V1 Administration* (SC31-6533)
- *APPC Application Suite V1 Programming* (SC31-6534)
- *APPC Application Suite V1 Online Product Library* (SK2T-2680)
- *APPC Application Suite Licensed Program Specifications* (GC31-6535)
- *z/OS V1R2.0 Communications Server: APPC Application Suite User's Guide* (SC31-8809)

Programming Publications

The following books contain information about programming:

- *Common Programming Interface Communications CPI-C Reference* (SC26-4399)
- *Communications Server for OS/2 Version 4 Application Programming Guide* (SC31-8152)

Other IBM Networking Publications

The following books contain information about other topics related to CS/AIX:

- *SDLC Concepts* (GA27-3093-04)
- *Local Area Network Concepts and Products: LAN Architecture* (SG24-4753-00)
- *Local Area Network Concepts and Products: LAN Adapters, Hubs and ATM* (SG24-4754-00)
- *Local Area Network Concepts and Products: Routers and Gateways* (SG24-4755-00)
- *Local Area Network Concepts and Products: LAN Operating Systems and Management* (SG24-4756-00)
- *IBM Network Control Program Resource Definition Guide* (SC30-3349)

Index

D

E

F

I

J

L

M

N

P

Q

R

S

T

U

W

Communicating Your Comments to IBM

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

Feel free to comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this document. However, the comments you send should pertain to only the information in this manual and the way in which the information is presented. To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Please send your comments to us in either of the following ways:

- If you prefer to send comments by FAX, use this number: 1+919-254-4028
- If you prefer to send comments electronically, use this address:
  - comsvrcf@us.ibm.com.
- If you prefer to send comments by post, use this address:

```
International Business Machines Corporation
Attn: z/OS Communications Server Information Development
P.O. Box 12195, 3039 Cornwallis Road
Department AKCA, Building 501
Research Triangle Park, North Carolina 27709-2195
```

Make sure to include the following in your note:

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

Program Number: 5765-E51

Printed in USA

SC31-8591-02