z/VM

CPI Communications User's Guide

version 6 release 1

SC24-6180-00

z/VM

CPI Communications User's Guide

version 6 release 1

SC24-6180-00

Note
Before using this information and the product it supports, read the general information under “Notices” on page 225.

This edition applies to version 6, release 1, modification 0 of IBM z/VM (product number 5741-A07) and to all subsequent releases and modifications until otherwise indicated in new editions.

This edition replaces SC24-6085-00.

Contents

Figures vii

Tables ix

About This Document xi
Intended Audience xi
Where to Find More Information xi

How to Send Your Comments to IBM xiii
If You Have a Technical Problem xiii

Chapter 1. Introduction 1
A Few Words about Our Format and Programs 1
Error Handling 1
The Programming Language Used for This Book 1
Before You Start 2
Setting Up the User IDs 2
Conventions Used in This Book 2
Pseudonyms 3
Visual Cues 3
CPI Communications Terms and Concepts for z/VM 4
Program Partners, Communications, and Resources 4
Like Using a Two-Way Radio 5
Type of Conversation to Be Used 6
Program Calls. 6
SAA CPI Communications Calls 6
z/VM Extensions to CPI Communications 7

Chapter 2. Starter Set CPI Communications Calls 9
Calls Used for Starting and Ending Conversations 9
Calls Used for Exchanging Data 9
Using the Starter Set Calls 9
Getting Started 9
Step 1. The Initialize_Conversation (CMINIT) Call 11
Step 2. The Allocate (CMALLC) Call 17
Step 3. The Send_Data (CMSEND) Call 20
Preparing the SERVR Virtual Machine 30
Step 4. The Accept_Conversation (CMACCP) Call 33
Step 5. The Receive (CMRCV) Call 36
Step 6. Adding a Receive (CMRCV) Loop to Our Requester Program 41
Step 7. Adding a Send_Data (CMSEND) Loop to Our Server 46
Step 8. The Deallocate (CMDEAL) Call 50
Summary with Flow Diagram 58
A Word about the Flow Diagrams 59
Flow Diagram for Starter Set Conversation 59

Chapter 3. Advanced CPI Communications Calls 63
Overview of Advanced CPI Communications Calls 63
Calls Used for Synchronization and Control 63
Using Advanced Set Calls 64
The Extract_Conversation_State (CMECS) Call 65
The Prepare_To_Receive (CMPTR) Call 70
The Set_Sync_Level (CMSSL) Call 74

The State Table–Finding Out Where You Can Go from Here 78
Confirmation Processing 80
The Confirm (CMCFM) Call 80
The Confirmed (CMCFMD) Call 81
The Set_Prepare_To_Receive_Type (CMSPTR) Call 88
The Set_Send_Type (CMSST) Call 93
The Set_Deallocate_Type (CMSDT) Call 98
The Extract_Conversation_Type (CMECT) Call 105
The Send_Error (CMSERR) Call 107
The Set_Conversation_Type (CMSCT) Call 109
The Set_Partner_LU_Name (CMSPLN) Call 113
The Set_TP_Name (CMSTPN) Call 116
Overviews of Additional Advanced Calls 119
Extract_Mode_Name (CMEMN) Call 119
Extract_Partner_LU_Name (CMEPLN) Call 119
Extract_Sync_Level (CMESL) Call 119
Request_To_Send (CMRTS) Call 119
Set_Error_Direction (CMSED) Call 120
Set_Fill (CMSF) Call 120
Set_Log_Data (CMSLD) Call 120
Set_Mode_Name (CMSMN) Call 120
Set_Return_Control (CMSRC) Call 121
Set_Receive_Type (CMSRT) Call 121
Test_Request_To_Send_Received (CMTRTS) Call 122
The Modified Sample Execs 122
The PROCESS Sample File Requester Exec 122
The SENDBACK Sample Server Exec 126
Summary 130

Chapter 4. VM Extensions to CPI Communications 131
The Relationship between VM and SAA CPI Communications 131
Overview of VM Extension Calls 131
Summary of VM Extension Calls 132
Managing a Resource 133
What Is a Resource Manager? 133
What Kinds of Resources Are There? 133
The Identify_Resource_Manager (XCIDRM) Call 135
The Terminate_Resource_Manager (XCTRRM) Call 139
The Wait_on_Event (XCWOE) Call 143
Security Considerations 156
The Set_Conversation_Security_Type (XCSCST) Call 158
The Set_Conversation_Security_User_ID (XCSCSU) Call 161
The Set_Conversation_Security_Password (XCSCSP) Call 162
Intermediate Servers 167
Setting Up the SERVR2 Virtual Machine 168
Converting the SERVR Virtual Machine into an Intermediate Server 170
Security Considerations for Intermediate Servers 183
The Extract_Conversation_Security_User_ID (XCECSU) Call 184
The Set_Client_Security_User_ID (XCSCUI) Call 191
Overview of Additional VM Extension Calls 198
Extract_Conversation_LUWID (XCECL) Call 198
Extract_Conversation_Workunitid (XCECWU) Call 198
Extract_Local_Fully_Qualified_LU_Name (XCELFQ) Call 198
Extract_Remote_Fully_Qualified_LU_Name (XCERFQ) Call 198
Extract_TP_Name (XCETPN) Call 198
Signal_User_Event (XCSUE) Call 198

The Completed Sample Execs. 199
The PROCESS Sample File Requester Exec 199
The SENDBACK Sample Intermediate Server Exec 203
The SENDSERV Sample Resource Manager Exec 210
Conclusion 215

Appendix A. Event Management for CPI Communications 217
The VMCPIC System Event. 217
Managing Events 218

Appendix B. CPI Communications Conversation States 223
Additional CPI Communications States 223

Notices 225
Programming Interface Information 227
Trademarks. 227

Glossary 229

Bibliography 231
Where to Get z/VM Information 231
z/VM Base Library 231
Overview 231
Installation, Migration, and Service 231
Planning and Administration. 231
Customization and Tuning 231
Operation and Use 231
Application Programming. 231
Diagnosis 232
z/VM Facilities and Features 232
Data Facility Storage Management Subsystem for VM 232
Directory Maintenance Facility for z/VM 232
Open Systems Adapter/Support Facility 232
Performance Toolkit for VM 233
RACF Security Server for z/VM 233
Remote Spooling Communications Subsystem Networking for z/VM 233
Prerequisite Products 233
Device Support Facilities 233
Environmental Record Editing and Printing Program. 233

Index 235

Figures

1. Partner Transaction Programs 4
2. A User Program Requests a Resource from a Resource Manager Program. 5
3. Output from PROCESS EXEC Showing Step 1 Results 14
4. Step 1 Output from PROCESS EXEC Showing Pseudonym 17
5. Output from PROCESS EXEC Showing Step 2 Results 19
6. Output from PROCESS EXEC Showing a Common Error 20
7. Output from PROCESS EXEC Showing Step 3 Results 24
8. Output from PROCESS EXEC after Adding UCOMDIR NAMES Entry 26
9. Output from PROCESS EXEC after SET COMDIR Command 28
10. System Response after Entering SET SERVER ON Command 28
11. Output from PROCESS EXEC after Entering the SET SERVER ON Command from the SERVR Console 29
12. Relationship between UCOMDIR and $SERVER$ NAMES Files 32
13. Output Resulting from Execution of SENDBACK EXEC. 35
14. Output from PROCESS EXEC Showing Step 4 Results 35
15. Output from SENDBACK EXEC Showing Step 4 Results 36
16. Output from PROCESS EXEC Showing Step 5 Results 40
17. Output from SENDBACK EXEC Showing Step 5 Results 41
18. Output from PROCESS EXEC Showing Step 6 Results 44
19. Output from SENDBACK EXEC Showing Step 6 Results 45
20. Output from PROCESS EXEC Showing Step 7 Results 48
21. Output from SENDBACK EXEC Showing Step 7 Results 49
22. Output from PROCESS EXEC Showing Step 8 Results 54
23. Output from SENDBACK EXEC Showing Step 8 Results 55
24. Flow Diagram for Starter Set Conversation 61
25. Results of First Two Calls from PROCESS EXEC 67
26. Results of Next Two Calls from PROCESS EXEC. 68
27. Results of Next Two Receive Calls from PROCESS EXEC 68
28. Completion of PROCESS EXEC Execution 68
29. Results of First Call from SENDBACK EXEC 69
30. Results of Next Two Calls from SENDBACK EXEC 69
31. Completion of SENDBACK EXEC Execution. 69
32. Execution Results after Adding CMPTR to PROCESS EXEC 73
33. Results from SENDBACK EXEC Execution 74
34. Results of Adding CMSSL Call to PROCESS EXEC 77
35. Results from SENDBACK EXEC Execution 77
36. Results of Confirmation Processing by PROCESS EXEC 86
37. Results of Confirmation Processing by SENDBACK EXEC 87
38. Results after Adding CMSPTR Call to PROCESS EXEC 92
39. Results of SENDBACK EXEC Execution 93
40. Results of PROCESS EXEC Execution 96
41. Results after Adding CMSST Call to SENDBACK EXEC 97
42. Results after Adding CMSDT Call to PROCESS EXEC 103
43. Results after Adding CMSDT Call to SENDBACK EXEC 103
44. Results of PROCESS EXEC Establishing a Basic Conversation 112
45. Results of SENDBACK EXEC Detecting a Basic Conversation 113
46. Results of Setting an Unknown LU Name from PROCESS EXEC 116
47. Results of Setting an Incorrect TP Name from PROCESS EXEC 118
48. Results on Server Virtual Machine Because of an Incorrect TP Name 119
49. Results of PROCESS EXEC Execution 142
50. SENDBACK EXEC Execution as a Resource Manager 143
51. Results of PROCESS EXEC Execution 151
52. Results of XCWOE to SENDBACK EXEC 152

53. Results of Starting SENDBACK EXEC on the SERVR User ID 156
54. Results of Entering QUIT at the SERVR Console 156
55. Results from PROCESS EXEC 165
56. Results from SENDBACK EXEC 166
57. SENDBACK Must Maintain Two Different Conversations 170
58. SENDBACK Assigns conversation_ID=ConvA and con_ID=ConvB 171
59. Results from PROCESS EXEC 179
60. Results from Intermediate Server's SENDBACK EXEC 180
61. Results from SERV2's SENDSERV EXEC 182
62. Requester's User ID Is Sent to VMUSR3 with TP-Model Application B's Allocate 183
63. Access Security User ID of Intermediate Server (VMUSR2) Sent to VMUSR3 183
64. Results from Requester's PROCESS EXEC 187
65. Results from Intermediate Server's SENDBACK EXEC 188
66. Results from SERVR2's SENDSERV EXEC 190
67. Results at SERVR2's Console 193
68. Results from Requester's PROCESS EXEC 194
69. Results from Server's SENDBACK EXEC 195
70. Results from SERVR2's SENDSERV EXEC 197

Tables

1. Overview of Communications Programs Using Starter Set Calls 9
2. Overview of Sample Programs with Advanced Set Calls 64
3. State Transitions for SENDBACK EXEC CPI Communications Calls 79
4. Overview of Sample Programs Using VM Extensions 134
5. Overview of Sample Intermediate Server Program 168
6. CPI Communications Conversation States 223
7. Additional Conversation States for Protected Conversations 223

About This Document

This document is intended to help you learn how to write communications programs. After working through this document, you should be able to use SAA Common Programming Interface (CPI) Communications routines to write communications programs that run in the CMS environment.

This document contains information on the IBM® z/VM® CPI Communications routines for application programmers.

- It provides an overview and CPI Communications as implemented in VM.
- It describes the starter set of SAA CPI Communications routines and builds a pair of simple communications programs using only these routines.
- It adds advanced SAA CPI Communications routines to programs to show additional functions available in SAA CPI Communications.
- It modifies programs to include several VM extension routines.

Intended Audience

You should read this document if you want to learn how to write communications programs, but are not familiar with CPI Communications. You do not need to have any experience with communications programming.

You should be knowledgeable about programming and familiar with CMS. The examples in this document are coded in REXX, but you do not need to be familiar with the language to work through this document. You can get acquainted with REXX by copying the examples.

Where to Find More Information

This book is designed to introduce you to CPI Communications in VM. These other books contain related information:

- *Common Programming Interface Communications Reference*, SC26-4399
- *z/VM: Connectivity*, SC24-6174
- *z/VM: CMS Application Development Guide*, SC24-6162
- *z/VM: REXX/VM Reference*, SC24-6221

Other books you may need to develop application programs are listed in the Bibliography of this book.

Links to Other Online Documents

If you are viewing the Adobe® Portable Document Format (PDF) version of this document, it might contain links to other documents. A link to another document is based on the name of the requested PDF file. The name of the PDF file for an IBM document is unique and identifies the edition. The links provided in this document are for the editions (PDF names) that were current when the PDF file for this document was generated. However, newer editions of some documents (with different PDF names) might exist. A link from this document to another document works only when both documents reside in the same directory.

How to Send Your Comments to IBM

We appreciate your input on this publication. Feel free to comment on the clarity, accuracy, and completeness of the information or give us any other feedback that you might have.

Use one of the following methods to send us your comments:

1. Send an e-mail to mhvrcfs@us.ibm.com
2. Visit the z/VM reader's comments Web page at www.ibm.com/systems/z/os/zvm/zvmforms/webqs.html
3. Mail the comments to the following address:
   IBM Corporation
   Attention: MHVRCFS Reader Comments
   Department H6MA, Mail Station P181
   2455 South Road
   Poughkeepsie, NY 12601-5400
   U.S.A.
4. Fax the comments to us as follows:
   From the United States and Canada: 1+845+432-9405
   From all other countries: Your international access code +1+845+432-9405

Include the following information:

- Your name and address
- Your e-mail address
- Your telephone or fax number
- The publication title and order number:
  z/VM V6R1 CPI Communications User's Guide
  SC24-6180-00
- The topic and page number related to your comment
- The text of your comment

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.

IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you submit to IBM.

If You Have a Technical Problem

Do not use the feedback methods listed above. Instead, do one of the following:

- Contact your IBM service representative.
- Contact IBM technical support.
- Visit the z/VM support Web page at www.vm.ibm.com/service/
- Visit the IBM mainframes support Web page at www.ibm.com/systems/support/z/

Chapter 1. Introduction

IBM's Systems Application Architecture® (SAA) has simplified the task of writing communications programs by providing the Communications element of the Common Programming Interface (CPI). CPI Communications, also called SAA communications interface, provides a programming interface for advanced program-to-program communications (APPC).

SAA CPI Communications defines a set of routines and parameters that are consistent across SAA environments. An application written using CPI Communications on z/VM can be transported to other SAA environments, provided the application does not use product-specific routines.

This book will help you get started writing simple communications programs using z/VM's implementation of CPI Communications. It can be viewed as an introduction and companion to the *Common Programming Interface Communications Reference*, which fully describes SAA CPI Communications and product-specific extensions.

A Few Words about Our Format and Programs

This book is designed as a self-study primer of the z/VM implementation of CPI Communications. Gradually, as you work through the book, we will build two sample CPI Communications programs. The format we will follow is to introduce a CPI Communications routine and provide a short discussion of its function. Then, we will add that routine to one of our programs and examine how it affects the results of the program.

After a few general routines have been added, one of our programs will request the contents of a particular file from the other program. In order to provide a broad introduction to CPI Communications, we will use most of the SAA and z/VM specfic routines in our programs. While some of these routines are not required for the particular function of this application, their use will help demonstrate how they may be used in other applications.

One of the more difficult parts of communications programming is determining what went wrong when a program does not work as expected. As we build the example programs in this book, things will not always work correctly the first time. The idea is to let you experience some of the common problems that can occur while we are here to help you through them. Having these experiences now may help you avoid, or at least recognize, similar problems in the future.

Error Handling

To limit the complexity of our example programs, we do not handle all error conditions. The degree of error handling necessary for your programs will depend on the application requirements.

The Programming Language Used for This Book

CPI Communications is intended for use by programs written in the SAA languages. We will be using REXX for the programs in this book because that is the one language we can be sure every z/VM system will have available (and besides, it is really easy to use). Familiarity with REXX would be helpful, but it is not essential to your CPI Communications education in this book. Just type carefully.

Before You Start

Before you begin working through this book, it is important to consider the following items:

- We expect you to know something about CMS and how to use an editor to create and modify a file. Familiarity with basic communications concepts would be helpful. If you have little or no experience in communications programming, it may be worthwhile for you to read the “Connectivity Programming in CMS” section of the *z/VM: CMS Application Development Guide*.
- A complete description of CPI Communications can be found in the *Common Programming Interface Communications Reference*. That book provides details concerning the SAA CPI Communications routines.
- We encourage you to implement our example programs, and to do so, you will need at least two z/VM user IDs and logon passwords (you will need three to complete the section on z/VM extensions) on a z/VM system. So that you can work with both of your virtual machines at the same time, you either should have access to a terminal or workstation that handles multiple sessions or should arrange to have two terminals at your desk.

Setting Up the User IDs

If you are not a system administrator, you will need to ask a system administrator or your supervisor to set up two or three virtual machines (user IDs) on the same z/VM system for you. Throughout this book, we refer to the virtual machines by the user IDs REQUESTR, SERVR, and SERVR2. You can use any user IDs you like. Just be careful to change the names appropriately as you copy the example programs. If you do **not** plan to work through the chapter on z/VM extensions to SAA CPI Communications, you can manage very well with only two user IDs (REQUESTR and SERVR).

We tested the example programs used in this book on virtual machines set up with 4MB of storage.

You will also need to ask the system administrator to add an IUCV ALLOW and an IPL CMS statement to the CP directory of the SERVR and SERVR2 virtual machines. The IUCV ALLOW statement authorizes the virtual machine to engage in communications. Without this statement, the server machine will not be able to function in the way that you need for this project. The IPL CMS statement automatically IPLs CMS when the user ID is logged on by the system as a result of a connection request.

If you plan to work through the z/VM extensions chapter, the SERVR virtual machine will also need Class B privilege so your program can set an alternate user ID. We will explain what this means when we need it.

Conventions Used in This Book

To make this book as usable as possible, we employ a couple of devices that are also used in the *Common Programming Interface Communications Reference*. Namely, we use pseudonyms for the various CPI Communications programming elements and we use different typefaces to establish visual cues. This section will help you understand and use these devices effectively.

Pseudonyms

To make it easier to follow what we are discussing and to improve the readability of this book, pseudonyms are used for the calls, characteristics, variables, and characteristic values that make up CPI Communications. For example, Initialize_Conversation is the pseudonym for the actual callable name CMINIT, and the pseudonym for one possible *return_code* value is CM_OK. Pseudonyms can also be used for integer values in program code by making use of equate or define statements.

Pseudonyms are not actually passed to CPI Communications as a string of characters. Instead, the pseudonyms represent integer values that are passed on the program calls. In the preceding example, CM_OK represents an integer value of 0. A mapping from valid pseudonyms to integer values can be found in an appendix of the *Common Programming Interface Communications Reference*.

z/VM provides sample pseudonym files called copy files for each of the SAA languages. See the *Common Programming Interface Communications Reference* (the “Programming Language Considerations” section and the “CPI Communications on VM/ESA CMS” appendix) for more details on these sample files.

Visual Cues

We use different typefaces to provide visual cues that should make working with this book easier. For terms that are being defined, we use boldface italics, like ***new term***. We show the example programs in a different typeface, as shown below. The first time program code is shown in the book, we show it in boldface. Whenever that particular program code appears again (as it does fairly often in the next chapter), we show it in the regular example typeface. This makes it easy to identify new sections of code and exactly where that code belongs in the program. Here is how the example typeface looks:

```
Let's say that this is code introduced previously.
This, on the other hand, is new code being inserted.
This makes it easy to see where to put the new code.
And now this is older code again.
```

The conventions we use for distinguishing the various programming elements are much the same as those used in the *Common Programming Interface Communications Reference*:

Call_Pseudonyms	Mixed case with underscores between words, for example, Initialize_Conversation
CALLS	All uppercase, for example, CMINIT
Variables	(This includes parameters and characteristics) mixed-case italics with underscores separating words, for example, *return_code*, *conversation_ID*
characteristic_values	All small uppercase letters with underscores between words, for example, CM_OK, CM_NO_DATA_RECEIVED
States	Boldface with the initial letter in uppercase and hyphens between words, for example, **Send-Pending**

When discussing the various communications calls, we generally list only the return codes that are pertinent to the sample program. For other possible return codes,

see the call descriptions in the *Common Programming Interface Communications Reference*. A complete list of possible return code pseudonyms and their descriptions can be found in the “Return Codes” appendix of that book. The return code pseudonyms and associated integer values can be found in the “Variables and Characteristics” appendix of that book.

CPI Communications Terms and Concepts for z/VM

Before we start looking at individual CPI Communications routines and begin writing our sample programs, let’s cover some background information. This section briefly summarizes some general terms and concepts related to communications programming and z/VM’s implementation of CPI Communications.

Program Partners, Communications, and Resources

Two CPI Communications programs exchange data using a ***conversation.*** The two programs involved in a conversation are called ***partners*** in the conversation, and each of these application programs can be referred to as a ***transaction program***. Figure 1 shows a conversation between Transaction Program A on UserID-1 and Transaction Program B on UserID-2.

Figure 1. Partner Transaction Programs

The terms local and remote are used to distinguish between the two sides of a conversation. If a program is being discussed as ***local***, its partner program is said to be the ***remote*** program for that conversation.

A common use of conversations is to access or modify resources. A ***resource*** can be a program, file, database, or any other entity that can be identified for application program processing. Common examples of resources in z/VM are a Shared File System (SFS) filepool or SQL/DS database.

The z/VM resources related to communications programming are classified as global, local, system, and private based on the scope under which they are identified for use. The only one we are concerned with in this book is the private resource. For information about global, local, and system resources, see the *z/VM: CMS Application Development Guide*. A ***private resource*** is identified only to the virtual machine in which it is located, although it can be accessed by authorized programs that reside anywhere in the same network.

A ***resource manager*** is a program that manages access to one or more z/VMresources. A resource manager gets requests from a user program to access resources owned by the resource manager, as shown in Figure 2 on page 5. The resource manager program runs in a ***server*** virtual machine. The private resource server virtual machine does not need to be logged on when a program requests to connect to the private resource manager. If the private server virtual machine is not logged on and its directory entry contains an IPL statement, CP will autolog it.

A ***user program*** is a transaction program that requests a service from a resource manager program. The user program, frequently called a ***requester program*** in this book, runs in a ***requester*** virtual machine.

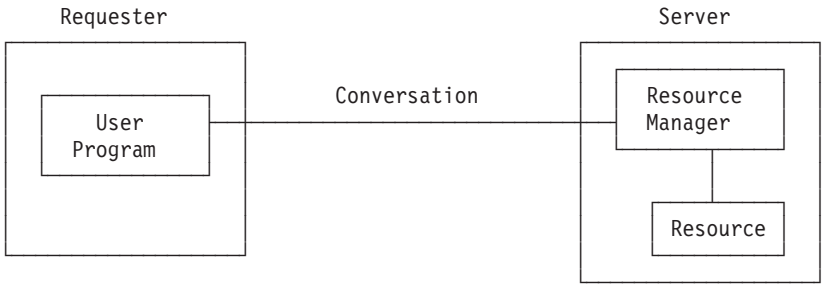


Figure 2. A User Program Requests a Resource from a Resource Manager Program

CPI Communications conversations use bidirectional ***half-duplex*** connections. Basically, this means that, although data can be sent by both partners, only one partner can send data at any given time.

Like Using a Two-Way Radio

The process of passing information back and forth between the partner programs, using a half-duplex conversation, can be compared to the way a pair of two-way radio operators communicate. Both situations require adherence to a set of communications protocols. The radios must be set to the same frequency; the virtual machines must be linked in some manner. Both the radio operator and the communications program initiating a conversation must specify the partner of the conversation. In general, the recipient of the information, another radio operator or another communications program, must acknowledge the request to communicate in order for an exchange to take place. The two partners in each type of conversation must take turns sending information to each other. Finally, one of the partners indicates to the other that the conversation is being terminated.

Protocols have been established in both realms (two-way radios and CPI Communications) to make communication easier and more efficient. For example, a two-way radio conversation might go something like this:

Radio ABC	**Radio XYZ**
ABC to XYZ, over	
	Go ahead, ABC, over
Give me your location, over	
	I'm at Broadway and Vine, over
Thank you, ABC out	
	XYZ out

Yes, it is rather crude, and even odd sounding to those of us who are accustomed to using the telephone for all our remote communications. But it serves a purpose. Fortunately, CPI Communications is rather more sophisticated than two-way radio communications, but the principle is still the same.

You will note that in our example, only one side (partner) talked at any one time. We can say that the talking (or sending) partner was in **Send** state and that the listening (or receiving) partner was in **Receive** state. These are the only two states available in two-way radio communications (unless you count idle and off as states). These are also the two basic states in the half-duplex protocol.

In CPI Communications, however, considerably more flexibility is possible even though we are still dealing with half-duplex connections. To use the flexibility provided by computer communications, new states were added on top of the basic half-duplex protocol. These states are a part of CPI Communications and they provide various capabilities to communications programs.

Even our simple two-way radio conversation suggests two more states, off (or no conversation) and start (or set up for a conversation, including idle time before the first transmission). Indeed, CPI Communications defines two states that coincide with these: **Reset** and **Initialize** states. **Reset** state means that there is no conversation activity and **Initialize** state means that a conversation is being set up. Thus, we already have four states in simple computer communications:

- **Reset**
- **Initialize**
- **Send**
- **Receive**

We will cover these and additional states as we work through the example programs in this book. For now, just remember that the type of communications protocol we are working with is basically like a two-way radio conversation. We can add a good deal of sophistication because we are also using computers and application programs to do the work for us.

Type of Conversation to Be Used

CPI Communications defines two types of conversations, mapped and basic. Mapped conversations allow programs to exchange arbitrary data records in formats agreed upon by the application programmers writing the communications programs. Basic conversations allow programs to exchange data in a standardized format, that is, a stream of data containing 2-byte length fields that specify the amount of data to follow before the next length field.

Because we will be using a mapped conversation and a private resource, the information discussed in this manual will focus on programming from that perspective. Basic conversations require much more work on the programmer's part than mapped conversations. You can find additional information on basic conversations in the *Common Programming Interface Communications Reference*.

Program Calls

CPI Communications programs communicate with each other by making program ***calls***. These calls are used to establish the characteristics of the conversation and to exchange data and control information between the programs.

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

SAA CPI Communications Calls

The SAA CPI Communications calls can be divided into two groups, a ***starter set*** and an ***advanced set***. The starter set consists of calls for starting and ending conversations and for exchanging data. Simple communications programs can be written exclusively with the routines in this group. Chapter 2, "Starter Set CPI Communications Calls," on page 9 uses only these communications routines to build the first working sample programs.

The advanced set consists of calls for synchronization and control, modifying conversation characteristics, and querying (extracting) conversation characteristics. These calls are used in Chapter 3, “Advanced CPI Communications Calls,” on page 63 to add more capabilities to our communications programs.

z/VM Extensions to CPI Communications

z/VM provides a group of ***extension calls*** to CPI Communications that allow programs to exploit z/VM’s capabilities. An application taking advantage of the added z/VM function, however, is not transportable to other SAA environments without modification. Chapter 4, “VM Extensions to CPI Communications,” on page 131 demonstrates the use of some of the routines in this group.

Chapter 2. Starter Set CPI Communications Calls

As we pointed out in Chapter 1, “Introduction,” the starter set of CPI Communications routines is made up of calls to start and end conversations and to exchange data. The six calls in this group are adequate for writing simple communications programs, and that is just what we are going to do with them momentarily. The following two tables will familiarize you with these calls.

Calls Used for Starting and Ending Conversations

Pseudonym	Call	Description	Page
Initialize_Conversation	CMINIT	Initializes values for various conversation characteristics before the conversation is allocated	11
Allocate	CMALLC	Establishes a conversation with a partner transaction program	17
Accept_Conversation	CMACCP	Accepts an incoming conversation	33
Deallocate	CMDEAL	Ends a conversation	50

Calls Used for Exchanging Data

Pseudonym	Call	Description	Page
Send_Data	CMSEND	Sends one data record to the remote program	20
Receive	CMRCV	Receives information from a given conversation	36

Using the Starter Set Calls

In this chapter you will learn how to use the starter set CPI Communications routines to start conversations, send data, receive data, and end conversations. The routines will be introduced in the order depicted in the following table. This table is not meant to illustrate the actual flows (transmission of information) of the conversation. A flow diagram is provided at the end of this chapter to help visualize how our communications programs work together.

Table 1. Overview of Communications Programs Using Starter Set Calls

Step	REQUESTR User ID	SERVR User ID
1.	**Initialize_Conversation**	
2.	**Allocate**	
3.	**Send_Data**	
4.		**Accept_Conversation**
5.		**Receive loop**
6.	**Receive loop**	
7.		**Send_Data loop**
8.	**Deallocate**	

Getting Started

Before you can make much progress in writing communications programs, you will need to set up two virtual machine user IDs. One of the virtual machines will be the

requester, and the other will act as a server. If you have not already obtained the user IDs, go back to “Setting Up the User IDs” on page 2 for a description of what you need. Log on the REQUESTR and SERVR user IDs after they are ready.

We will begin working from the REQUESTR user ID, where we will create a user program called PROCESS EXEC.

FYI: REXX Considerations

Because this is a REXX exec, we will want to begin with a comment line (`/* */`).

In REXX you can specify a subcommand environment to help ensure predictable and efficient execution of REXX execs. The REXX subcommand environment for CPI Communications is called CPICOMM, and we can either specify `ADDRESS CPICOMM` once for the entire exec or each time we issue a call. Because our focus is on CPI Communications, we will go ahead and set CPICOMM as the primary subcommand environment at the top of the program. As a result, we need to keep in mind that the appropriate `ADDRESS` statement will need to precede any CMS or CP commands issued by our program.

As part of our basic error checking, we want our program to check the REXX special variable RC. If RC is not zero, then the CPI Communications call was not invoked and none of the call’s output parameters contain valid values. The chapter on invoking communications routines in the *z/VM: REXX/VM Reference* lists possible values for the RC variable. We will add the REXX instruction `SIGNAL ON ERROR` to our program, which will result in the RC variable being monitored for us. If RC is set to a nonzero value, a subroutine named Error in our program will be called automatically.

These REXX considerations suggest that we start our program with the following lines:

```
/*====================================================================*/
/*  PROCESS EXEC - Sample file requester application.                 */
/*====================================================================*/

/*--------------------------------------------------------------------*/
/* Set up REXX environment for program-to-program communications      */
/* and enable trapping of REXX errors.                                */
/*--------------------------------------------------------------------*/
address cpicomm
signal on error

GetOut:
  exit

/*------------------------- Subroutines --------------------------*/

Error:
/*--------------------------------------------------------------------*/
/* Report error when REXX special variable RC is not 0.               */
/*--------------------------------------------------------------------*/
say
say '* ERROR:  REXX has detected an error'
say '          The return code variable RC was set to' rc
signal GetOut
```

Now, we are ready to begin our exploration of CPI Communications.

Step 1. The Initialize_Conversation (CMINIT) Call

Before communication can begin, we must initiate a conversation. There are two steps involved in initiating a CPI Communications conversation: initialization of conversation characteristics and allocation of the conversation. The purpose of the Initialize_Conversation (CMINIT) routine is to handle the first step of that process.

Conversation characteristics define the conversation to be established. Returning to the two-way radio conversation for an example, the radio operator must decide what transmission frequency to use before transmitting any information. So, the transmission frequency can be considered a characteristic of the radio conversation.

Various characteristics are associated with communications conversations, and the Initialize_Conversation call initializes most of those characteristics to predefined values. (The “Conversation Characteristics” section in the *Common Programming Interface Communications Reference* contains a complete list of these conversation characteristics and their default values. For example, the *conversation_type* characteristic is initialized to CM_MAPPED_CONVERSATION.) The default characteristic settings are sufficient in many cases, but if any of these default values are not appropriate for a particular application, they can be changed easily with calls to other CPI Communications routines that we will discuss in the next chapter.

Another result of a call to the Initialize_Conversation routine is that CPI Communications side information is examined. The values of several additional conversation characteristics are set based on the contents of the side information.

FYI: Side Information

Certain information about the partner program that will be participating in a conversation must be provided to complete the initialization of conversation characteristics. This data is referred to as side information and it identifies the location of the partner program.

VM’s implementation of side information uses CMS communications directory files. A communications directory file is a NAMES file that can be set up either on a system level (by a system administrator) or on a user level. We will discuss this in more detail later.

During the discussion of a communications routine, we will want to examine the particular parameters associated with it. When showing the general call format for a routine, we will indicate which parameters require that a value be provided on the call (input parameters) and which ones return a value upon completion of the call (output parameters). The general format for calling Initialize_Conversation is:

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

You will notice that this example is similar to the call format shown for Initialize_Conversation in the *Common Programming Interface Communications Reference*, except that we are emphasizing whether a parameter serves as input or output. Note that REXX uses a slightly different call format; we are using the

standard format just for illustration here. CALL is the language-specific syntax for calling CPI Communications routines. CMINIT is the name of this call.

Input Parameter

The ***sym_dest_name*** parameter is the sole input parameter on this call, so it is the only one for which we need to provide a value. *Sym_dest_name* is an abbreviation of “symbolic destination name”, which is meant to indicate that this parameter identifies the remote partner for the conversation. The value provided as input to the routine is used as an index into a side information file. Data from the corresponding entry in side information initializes several conversation characteristics.

Output Parameters

The two output parameters for Initialize_Conversation are *conversation_ID* and *return_code*. The ***conversation_ID*** is assigned by CPI Communications to uniquely identify this conversation. It is used as input on all subsequent CPI Communications calls made for the conversation.

Each CPI Communications routine has a ***return_code*** parameter, which specifies the result of the call execution. All of the return codes are defined in the *Common Programming Interface Communications Reference*, although the *return_code* parameter for a particular routine can only take on a subset of those values.

It is beyond the intent of this tutorial to cover all of the *return_code* values that are possible for each routine. Instead, we are going to highlight just some of the more common ones. For Initialize_Conversation, there are two codes of interest:

CM_OK (0)
: indicates that the characteristics for a conversation have been successfully initialized. That’s the code we will want to get back.

CM_PRODUCT_SPECIFIC_ERROR (20)
: indicates that a CMS error has occurred. Whenever a CM_PRODUCT_SPECIFIC_ERROR is returned, a file on your A-disk called CPICOMM LOGDATA is updated with a message to help you understand what type of problem was encountered. Message lines are appended to this file, so the last line in the file is the most recent. Additional information on product specific errors can be found in the Appendix “CPI Communications on VM/ESA CMS” in the *Common Programming Interface Communications Reference*.

Results of the Call

If the Initialize_Conversation call completes successfully, meaning that the *return_code* was CM_OK, then the conversation enters **Initialize** state. Prior to this point, we had been in **Reset** state. Keep this in mind. It will be useful later on when we discuss states in more detail.

Adding CMINIT to Our Requester Program

To call the Initialize_Conversation routine, we will need to use the routine’s callable name CMINIT.

As previously noted, the *sym_dest_name* is the only input parameter for this routine. We can either assign a value (the identifier for our remote partner) to the *sym_dest_name* variable in our program before calling Initialize_Conversation, or we can accept the value from the console as an operand on the exec call. To make the program more dynamic, we will provide the *sym_dest_name* as console input. We will need to add an `ARG sym_dest_name` statement to our REXX exec to pull the symbolic destination name from the console and place it in the program stack.

(Ordinarily, we would also add some error checking to ensure that any input from the console meets our expectations, but we will forgo that in these simple examples to save typing and focus on the communications calls.)

In addition, we will display the results of the Initialize_Conversation call. We will add a `say` statement to show which call was executed. By passing a list of parameters to a subroutine called TraceParms, we can also display the current value for each of the parameters.

We will be checking the *return_code* output parameter after each CPI Communications call to determine whether the call completed successfully. If *return_code* contains a nonzero value, we will want to know about it, so we will add a subroutine called ErrorHandler, which will be called whenever the program encounters a problem.

FYI: More REXX Considerations

Note that when we add a call to a CPI Communications routine, we need to put single quotation marks around the call.

The subroutine TraceParms uses the REXX function WORDS to determine the number of blank-delimited words in a string. In our case, each word is a parameter name, so WORDS will return the number of parameters that need to be processed. TraceParms also uses the REXX functions WORD, which returns the specified word in a string, and VALUE, which returns the value that a specified symbol represents. So, we can use WORD to extract a parameter name followed by VALUE to display the current value of that parameter.

Adding the Initialize_Conversation call and the new subroutines to the PROCESS EXEC results in these changes (denoted by highlighting):

```
/*====================================================================*/
/*  PROCESS EXEC - Sample file requester application.                 */
/*====================================================================*/

arg sym_dest_name .                              /* get user's input */
/*--------------------------------------------------------------------*/
/* Set up REXX environment for program-to-program communications      */
/* and enable error trapping of REXX errors.                          */
/*--------------------------------------------------------------------*/
address cpicomm
signal on error
/*--------------------------------------------------------------------*/
/* Initialize the conversation.                                       */
/*--------------------------------------------------------------------*/
'CMINIT conversation_ID sym_dest_name return_code'
say; say 'Routine called:  CMINIT'
if (return_code ¬= 0) then call ErrorHandler 'CMINIT'
call TraceParms 'conversation_ID sym_dest_name return_code'

GetOut:
  exit

/*------------------------- Subroutines --------------------------*/

TraceParms:
/*--------------------------------------------------------------------*/
/* Display parameters and their values as passed to this subroutine.*/
/*--------------------------------------------------------------------*/
parse arg parmlist
```

```
do word_num = 1 to words(parmlist)
  parameter = word(parmlist,word_num)
  select
    when (parameter = 'return_code') then
      say '  return_code is' return_code
    otherwise
      say ' ' parameter 'is' value(parameter)
  end
end

return


Error:
/*--------------------------------------------------------------------*/
/* Report error when REXX special variable RC is not 0.               */
/*--------------------------------------------------------------------*/
say
say '* ERROR:  REXX has detected an error'
say '          The return code variable RC was set to' rc
signal GetOut


ErrorHandler:
/*--------------------------------------------------------------------*/
/* Report routine that failed and the error return code.              */
/*--------------------------------------------------------------------*/
parse arg routine_name
say
say '* ERROR:  An error occurred during a' routine_name 'call'
say '          The return_code was set to' return_code
signal GetOut
```

Throughout the book, we will continue using highlighting as in this example to designate new or changed code whenever we update one of our execs.

Please type very carefully each time you add new code to the program and check for typing mistakes because these can cause errors. Now file the exec and run it. Using GETFILE as the *sym_dest_name*, enter

```
process getfile
```

at the command line.

The following should be displayed on your terminal:

```
process getfile

Routine called:  CMINIT
  conversation_ID is 00000000
  sym_dest_name is GETFILE
  return_code is 0
Ready;
```

Figure 3. Output from PROCESS EXEC Showing Step 1 Results

As you can see, our return code is zero. Therefore, it should be safe to assume that the conversation initialization was a success. The symbolic destination name (*sym_dest_name*) is, of course, the value we provided when the exec was invoked.

The *conversation_ID* has special significance. It uniquely defines the local program's side of the conversation for CPI Communications.

Did you notice that we compared the *return_code* value to zero after the Initialize_Conversation call? We could add more meaning to the program by substituting a pseudonym for the integer zero. Pseudonyms are defined for the integer values of various CPI Communications variables and characteristics and can be found in an appendix of the *Common Programming Interface Communications Reference*.

If you look up *return_code* in the variables and characteristics appendix mentioned above, you will see that the pseudonym associated with a *return_code* of zero is CM_OK. So, we can substitute the pseudonym CM_OK in our program to replace references to a *return_code* value of zero. But, while CM_OK is equivalent to zero, it is a pseudonym that only has meaning as a value for the *return_code* parameter. For example, CM_WHEN_SESSION_ALLOCATED is the pseudonym that should be used to refer to a *return_control* value of zero. Although both pseudonyms represent values of zero, the pseudonyms themselves provide more specific information about the value with regard to the variable or characteristic they are associated with.

FYI: Copy Files—the Easy Way to Use Pseudonyms

CMREXX COPY is a sample CPI Communications pseudonym file that includes all of the conversation characteristic values. It equates pseudonyms to their actual integer values. CMREXX COPY should be located on your system disk.

Before we can use the pseudonyms in our program, they have to be defined to our program. We can do this by reading the CMREXX COPY file values into storage using the EXECIO command with a DO loop that includes the REXX `interpret` statement.

We can also take advantage of REXX's compound symbol support, which permits arbitrary indexing of collections of variables that have a common stem. For example, by using the *return_code* value returned from Initialize_Conversation as an index to a compound variable called *cm_return_code*, we can display a pseudonym result rather than just an integer. CMREXX COPY includes compound variable versions of each of the conversation characteristics.

The "Programming Language Considerations" section of the *Common Programming Interface Communications Reference* contains information on pseudonym files for other SAA languages.

Let's try using the pseudonyms in our program. First, we will add an EXECIO statement to process the CMREXX COPY file, and then we will use pseudonyms wherever we can.

```
/*====================================================================*/
/*  PROCESS EXEC - Sample file requester application.                 */
/*====================================================================*/

arg sym_dest_name .                              /* get user's input */
/*--------------------------------------------------------------------*/
/* Set up REXX environment for program-to-program communications    */
/* and enable error trapping of REXX errors.                          */
/*--------------------------------------------------------------------*/
address cpicomm
signal on error
/*--------------------------------------------------------------------*/
```

```
/* Equate pseudonyms to their integer values based on the           */
/* definitions contained in the CMREXX COPY file.                   */
/*------------------------------------------------------------------*/
address command 'EXECIO * DISKR CMREXX COPY * (FINIS STEM PSEUDONYM.'
do index = 1 to pseudonym.0
  interpret pseudonym.index
end
/*------------------------------------------------------------------*/
/* Initialize the conversation.                                     */
/*------------------------------------------------------------------*/
'CMINIT conversation_ID sym_dest_name return_code'
say; say 'Routine called:  CMINIT'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMINIT'
call TraceParms 'conversation_ID sym_dest_name return_code'

GetOut:
  exit

/*------------------------- Subroutines --------------------------*/

TraceParms:
/*------------------------------------------------------------------*/
/* Display parameters and their values as passed to this subroutine.*/
/*------------------------------------------------------------------*/
parse arg parmlist
do word_num = 1 to words(parmlist)
  parameter = word(parmlist,word_num)
  select
    when (parameter = 'return_code') then
      say '  return_code is' cm_return_code.return_code
    otherwise
      say ' ' parameter 'is' value(parameter)
  end
end

return


Error:
/*------------------------------------------------------------------*/
/* Report error when REXX special variable RC is not 0.             */
/*------------------------------------------------------------------*/
say
say '* ERROR:  REXX has detected an error'
say '          The return code variable RC was set to' rc
signal GetOut


ErrorHandler:
/*------------------------------------------------------------------*/
/* Report routine that failed and the error return code.            */
/*------------------------------------------------------------------*/
parse arg routine_name
say
say '* ERROR:  An error occurred during a' routine_name 'call'
say '          The return_code was set to' cm_return_code.return_code
signal GetOut
```

After making the updates, file the exec and start it again with

```
process getfile
```

Now, your results should look like:

```
process getfile

Routine called:  CMINIT
  conversation_ID is 00000000
  sym_dest_name is GETFILE
  return_code is CM_OK
Ready;
```

Figure 4. Step 1 Output from PROCESS EXEC Showing Pseudonym

The *return_code* parameter was again set to zero following the completion of the Initialize_Conversation call. Our program then displayed the value of *cm_return_code.return_code*, or *cm_return_code*.0, which is CM_OK.

From this point on, your particular *conversation_ID* value may differ from the one we show. Differences in this value do not matter, as long as you use the value returned on the Initialize_Conversation call when issuing other calls on this conversation.

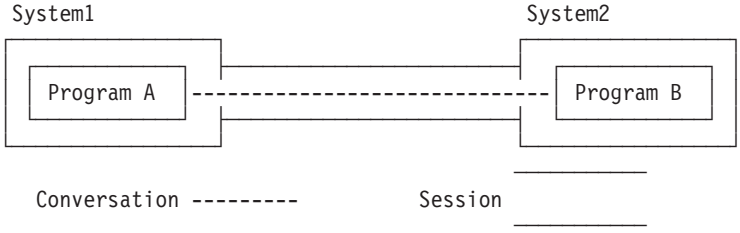
If you are not fond of typing, you might want to make a copy of PROCESS EXEC and name it SENDBACK EXEC. Be sure to change the comment line at the top of the SENDBACK copy to reflect its name and description, remove the "`arg sym_dest_name .`" line, and remove the seven lines comprising the "Initialize the conversation" section. We will use this file later.

Step 2. The Allocate (CMALLC) Call

Having initialized a conversation (using the Initialize_Conversation call), an application uses the Allocate (CMALLC) call to establish the conversation with its partner transaction program.

FYI: What the Allocate Call Actually Does

Before the Allocate call can establish a conversation, it must make sure that there is a logical connection between the local program's system, also known as a logical unit (LU) in this context, and the remote program's system. This logical connection is called a **session**, shown in the following drawing as a pair of solid lines between the System1 LU and the System2 LU.



Each session can support one conversation. In our sample programs this is not really important, because both partners are considered to be on virtual machines on the same VM/ESA® system and are therefore in the same LU. Sessions are required on VM only when connecting through an SNA network. A return code of CM_OK indicates that the session was successfully established. However, an active session (successful execution of Allocate) does not guarantee that the communication partner's transaction program can be started.

The allocation request may not be sent until the local send buffer becomes full or is flushed. We will discuss this in more detail in the next chapter.

The format for Allocate is:

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

Input Parameter

Use the ***conversation_ID*** parameter to identify the conversation.

Output Parameter

The values for ***return_code*** on an Allocate call that are of interest to us are:

CM_OK (0)
: indicates that the conversation has been allocated and that the local program has entered **Send** state.

CM_PARAMETER_ERROR (19)
: indicates that there is a problem with one of the target destination characteristics provided in side information or explicitly set.

CM_PRODUCT_SPECIFIC_ERROR (20)
: indicates that an error unique to the VM product has occurred. Check the CPICOMM LOGDATA file for a summary of the error.

CM_PROGRAM_PARAMETER_CHECK (24)
: indicates that the specified conversation ID is unassigned.

CM_PROGRAM_STATE_CHECK (25)
: indicates that the conversation is not in **Initialize** state, meaning the specified conversation either has not been initialized or has already been allocated.

Results of the Call

When *return_code* indicates CM_OK, the conversation enters **Send** state. This does not, however, guarantee that the partner's transaction program has been started.

Adding CMALLC to Our Requester Program

Let's add the Allocate call to the PROCESS EXEC. To save space, we will not show the entire program each time we add a new routine. So that you can determine where the new code should be inserted, we will show only the sections of code that precede and follow it. Your exec now should have the following lines in it:

```
/*================================================================*/
/*  PROCESS EXEC - Sample file requester application.             */
/*================================================================*/

.
.
.
/*----------------------------------------------------------------*/
/* Initialize the conversation.                                   */
/*----------------------------------------------------------------*/
'CMINIT conversation_ID sym_dest_name return_code'
say; say 'Routine called:  CMINIT'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMINIT'
call TraceParms 'conversation_ID sym_dest_name return_code'
/*----------------------------------------------------------------*/
/* Allocate the conversation.                                     */
/*----------------------------------------------------------------*/
'CMALLC conversation_ID return_code'
say; say 'Routine called:  CMALLC'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMALLC'
call TraceParms 'conversation_ID return_code'
```

```
GetOut:
  exit

/*------------------------- Subroutines ---------------------------*/

   ⋮
```

Now file the exec and execute it, again using GETFILE as the *sym_dest_name*:

```
process getfile
```

The results should be:

```
process getfile

Routine called:  CMINIT
  conversation_ID is 00000000
  sym_dest_name is GETFILE
  return_code is CM_OK

Routine called:  CMALLC
  conversation_ID is 00000000
  return_code is CM_OK
Ready;
```

Figure 5. Output from PROCESS EXEC Showing Step 2 Results

As you can see, both return codes were CM_OK, so the conversation initialization and allocation appear to have been successful and we are now in **Send** state.

FYI: If You Got a Product-Specific Error

If, instead of the output shown in Figure 5, you received a CM_PRODUCT_SPECIFIC_ERROR, look in the CPICOMM LOGDATA A file for the following message:

```
CMALLC_PRODUCT_SPECIFIC_ERROR:  CMSIUCV CONNECT completed with return
   code 1011
```

As an exercise, you may want to track down this message to see if you can find out what it means. (Hint: You will need the *z/VM: CMS Macros and Functions Reference* to look up the CMSIUCV macro and the *z/VM: CP Programming Services* book to look up the APPCVM macro.) This error is most likely saying that your VM system does not have the Transparent Services Access Facility (TSAF) installed. TSAF is used for communications among up to eight VM systems.

This problem will be resolved before we get to Step 4, so for now, please continue reading and adding code to the program, but do not try to run the program because your results will differ from those shown.

Note on a Common Error

Let's take a moment to look at an error that can show up rather easily in a program.

The Allocate call expects to have a conversation ID passed to it in a valid format. Suppose we misspell our parameter name when adding the Allocate call to our program, like this:

```
'CMALLC onversation_ID return_code'
```

While we can choose any parameter name that we want, our program will be incorrect because the conversation identifier was previously stored in *conversation_ID*, not *onversation_ID*. Here is the output displayed when this altered program is executed:

```
process getfile

Routine called:  CMINIT
  conversation_ID is 00000000
  sym_dest_name is GETFILE
  return_code is CM_OK
DMSAXR1292E Error calling CPI-Communications routine, return code -26002
    30 *-* 'CMALLC onversation_ID return_code'
       +++ RC(-26002) +++

* ERROR:  REXX has detected an error
          The return code variable RC was set to -26002
Ready;
```

Figure 6. Output from PROCESS EXEC Showing a Common Error

Error codes that can be returned from ADDRESS CPICOMM are documented in the *z/VM: REXX/VM Reference* in the chapter called "Invoking Communications Routines". The explanation listed for -26*nnn* indicates that there is a problem with parameter number *nnn*. The call name, for example CMINIT, is considered to be the first parameter. In our case, *nnn* is 002 indicating that the second parameter is causing the problem. By examining that parameter in our program, we see that the error can be corrected quite easily.

Step 3. The Send_Data (CMSEND) Call

The Send_Data (CMSEND) call sends up to 32767 bytes of data to the remote program. When issued during a mapped conversation (which we are using), this call sends one data record to the remote program. In this context, a data record is the contents of the buffer passed on the Send_Data call.

The format for Send_Data is:

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

Input Parameters

The Send_Data call expects three input parameters, including the ***conversation_ID***. The ***buffer*** parameter specifies the data record to be sent. While this record can be defined within the program, it may be useful for some applications to set *buffer* to a character string provided by console input.

The ***send_length*** parameter specifies the size of the *buffer* contents in bytes (up to 32767).

Output Parameters

The ***request_to_send_received*** parameter returns an indication of whether a request-to-send notification has been received from the partner program. The *request_to_send_received* variable can have the following values:

CM_REQ_TO_SEND_NOT_RECEIVED (0)

CM_REQ_TO_SEND_RECEIVED (1)

If a request-to-send notification was received, it means that the remote program has requested that the local program's end of the conversation enter **Receive** state, which would place the remote program's end of the conversation in **Send** state.

Request_to_send_received does not return a value when *return_code* is either CM_PROGRAM_PARAMETER_CHECK or CM_PROGRAM_STATE_CHECK.

Some of the ***return_code*** values of interest to us are:

CM_OK (0)
: indicates that the Send_Data call executed successfully.

CM_SECURITY_NOT_VALID (6)
: indicates that the allocation request was rejected by the remote LU because the access security information provided by the local system is invalid. The VM appendix to the *Common Programming Interface Communications Reference* lists more specific possible explanations for this return code. The conversation is in **Reset** state.

CM_TPN_NOT_RECOGNIZED (9)
: indicates that the Allocation request was rejected by the remote LU because the specified remote program name was not recognized at the remote system. The VM appendix to the *Common Programming Interface Communications Reference* lists more specific possible explanations for this return code. The conversation is in **Reset** state.

CM_TP_NOT_AVAILABLE_NO_RETRY (10)
: indicates that the allocation request was rejected because the remote system could not start the remote program. The condition is not temporary so the program should not retry the allocation. The conversation is in **Reset** state.

CM_TP_NOT_AVAILABLE_RETRY (11)
: indicates that the allocation request was rejected because the remote system could not start the remote program. The condition may be temporary so the program can retry the allocation. The conversation is in **Reset** state.

CM_DEALLOCATED_ABEND (17)
: indicates that the remote program or system deallocated the conversation, terminated abnormally, or ended without deallocating the conversation. The conversation is in **Reset** state.

CM_PRODUCT_SPECIFIC_ERROR (20)
: indicates that an error unique to the VM product has occurred. Check the CPICOMM LOGDATA file for a summary of the error.

CM_PROGRAM_PARAMETER_CHECK (24)
: indicates that the specified conversation ID is unassigned or that the *send_length* is greater than 32767.

CM_PROGRAM_STATE_CHECK (25)
: most commonly indicates that the conversation is not in **Send** or **Send-Pending** state.

CM_RESOURCE_FAILURE_NO_RETRY (26)
: indicates that a failure occurred that caused the conversation to be terminated prematurely or the remote program ended without deallocating the conversation. The VM appendix to the *Common Programming Interface Communications Reference* lists more specific possible explanations for this return code. The condition is not temporary. The conversation is in **Reset** state.

CM_RESOURCE_FAILURE_RETRY (27)
: indicates that a failure occurred that caused the conversation to be

terminated prematurely. This could occur if the TSAF virtual machine encountered a problem during its processing or if the TSAF link went down. The condition may be temporary so the program can retry the allocation request. The conversation is in **Reset** state.

FYI: A Note on When Errors Are Reported

It is worth mentioning that some CPI Communications errors are not reported when they first occur. If you read through the possible *return_code* values listed, you probably noticed that some of them seemed more appropriate to an Allocate call. In fact, they are indeed allocation errors—they are reported because the Allocate call did not result in a conversation for one reason or another. However, they are not actually reported until some call following the Allocate, such as a Send_Data (CMSEND) call, is executed.

dhis delay in reporting errors should be kept in mind while debugging application errors. Your program should be prepared to handle allocation errors on other calls, such as Send_Data and Receive (CMRCV).

Results of the Call

A number of factors, including the values of various conversation characteristics, can affect the results of the Send_Data call. We will discuss some of these later in the book. For now, a CM_OK *return_code* value indicates that the data record has been "sent" and that the conversation is still in **Send** state.

Adding CMSEND to Our Requester Program

Let's add the Send_Data call to the PROCESS EXEC.

Since this is a program to request the contents of a file, we will provide the file name, file type, and file mode of the file we are requesting when we invoke the exec. By placing the file name, file type, and file mode in the *buffer* parameter, this information can be passed to the server on the Send_Data call.

The existing `ARG` statement needs to be updated to retrieve the additional arguments we will be providing. We will also need to initialize the *buffer* and *send_length* variables.

Your exec should now have the following lines in it:

```
/*====================================================================*/
/*  PROCESS EXEC - Sample file requester application.                 */
/*====================================================================*/

arg sym_dest_name fname ftype fmode .           /* get user's input */
/*--------------------------------------------------------------------*/
/* If a file was not specifically requested, set up a default.        */
/*--------------------------------------------------------------------*/
if (fname = '') then
  do
    fname = 'TEST'
    ftype = 'FILE'
    fmode = 'A'
  end
say 'Requesting the file: ' fname ftype fmode
:
:
/*--------------------------------------------------------------------*/
/* Initialize the conversation.                                       */
/*--------------------------------------------------------------------*/
```

```
'CMINIT conversation_ID sym_dest_name return_code'
say; say 'Routine called:  CMINIT'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMINIT'
call TraceParms 'conversation_ID sym_dest_name return_code'
/*-------------------------------------------------------------------*/
/* Allocate the conversation.                                        */
/*-------------------------------------------------------------------*/
'CMALLC conversation_ID return_code'
say; say 'Routine called:  CMALLC'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMALLC'
call TraceParms 'conversation_ID return_code'
/*-------------------------------------------------------------------*/
/* Send the name of the file being requested to the partner program.*/
/*-------------------------------------------------------------------*/
buffer = fname ftype fmode
send_length = length(buffer)
'CMSEND conversation_ID buffer send_length',
       'request_to_send_received return_code'
say; say 'Routine called:  CMSEND'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMSEND'
call TraceParms 'conversation_ID buffer send_length',
                'request_to_send_received return_code'

GetOut:
  exit

/*------------------------- Subroutines --------------------------*/

TraceParms:
/*-------------------------------------------------------------------*/
/* Display parameters and their values as passed to this subroutine.*/
/*-------------------------------------------------------------------*/
parse arg parmlist
do word_num = 1 to words(parmlist)
  parameter = word(parmlist,word_num)
  select
    when (parameter = 'return_code') then
      say '  return_code is' cm_return_code.return_code
    when (parameter = 'buffer') then
      say '  buffer is' left(buffer,send_length)
    when (parameter = 'request_to_send_received') then
      say '  request_to_send_received is',
          cm_request_to_send_received.request_to_send_received
    otherwise
      say ' ' parameter 'is' value(parameter)
  end
end

return
```

⋮

FYI: The LEFT Function in REXX

The REXX function LEFT returns the leftmost characters of a character string for a specified length. Specifying the *send_length* with the LEFT function of REXX ensures that the `say` statement will display exactly what we put into the *buffer*.

File the exec and run it. This time, in addition to using GETFILE as the *sym_dest_name*, use TEST as the *fname*, FILE as the *ftype*, and A as the *fmode*. (Having included this file name as a default in our program, we do not have to

specify it each time we invoke the exec in the future.) The fact that we have not created TEST FILE A does not matter at this point.

After starting the exec with

```
process getfile test file a
```

the following should be displayed:

```
process getfile test file a
Requesting the file:  TEST FILE A

Routine called:  CMINIT
  conversation_ID is 00000000
  sym_dest_name is GETFILE
  return_code is CM_OK

Routine called:  CMALLC
  conversation_ID is 00000000
  return_code is CM_OK

Routine called:  CMSEND

* ERROR:  An error occurred during a CMSEND call
          The return_code was set to CM_TPN_NOT_RECOGNIZED
Ready;
```

Figure 7. Output from PROCESS EXEC Showing Step 3 Results

The return code of CM_TPN_NOT_RECOGNIZED indicates a problem. It looks like our Initialize_Conversation and Allocate calls were successful, but the Send_Data call did not work. If we look back at the possible values for the *return_code* parameter on the Send_Data call, we see that we did not actually have a conversation even though the Allocate call completed with a *return_code* value of CM_OK. This allocation error is what the FYI box 22 was referring to.

So the allocation request was rejected because the program name specified on the Initialize_Conversation call was not recognized. The problem is that the symbolic destination name GETFILE has not been identified as the transaction program name (*TP_name*) of a private resource.

Okay, then how can we define a symbolic destination name? Data about the target destination location is typically placed in side information, which is a CMS communications directory in VM. The communications directory is a good place to check first when a return code of CM_TPN_NOT_RECOGNIZED is received. Based on data in the communications directory, the symbolic destination name is resolved and the *TP_name* is initialized for use in the allocation request to the partner program.

FYI: CMS Communications Directories

Let's digress from our program for a moment and investigate how VM implements side information.

VM implements side information with CMS communications directory files. A communications directory file is a special CMS NAMES file. Communications directories can be set up at either a system or a user level.

System Communications Directory: A system administrator sets up the system-level communications directory. The default name defined in the system profile exec (SYSPROF EXEC) for this communications directory is **SCOMDIR NAMES**. It is usually located on the system S-disk or in a public SFS file pool where all users can read it.

User Communications Directory: Any CMS user can create a personal CMS communications directory. **UCOMDIR NAMES** is the default name defined in the system profile exec (SYSPROF EXEC) for the user-level directory. In general, this directory is only necessary if an application uses symbolic destination names that are not already in the SCOMDIR NAMES file or if there is a need to override the system-defined values.

Note: You can create or change your communications directories using the NAMES command with the COMDIR option. See the NAMES command usage notes in the *z/VM: CMS Commands and Utilities Reference* for more information.

When VM resolves a symbolic destination name, the user-level directory, if one exists, is checked first for a matching entry. If the user-level communications directory does not contain the specified symbolic destination name, CMS searches the system-level communications directory for a matching entry. If a symbolic destination name is defined in both the UCOMDIR NAMES file and the SCOMDIR NAMES file, only the information in the UCOMDIR NAMES file is used.

If the resource identified in the initialization request does not match a symbolic destination name defined in either of the CMS communications directories, then the initialization request is processed using the specified symbolic destination name as the name of a global or local resource located in the same TSAF collection as the user program.

Here are the communications directory tags and associated values we will be using now:

Tag	What Value the Tag Specifies
`:nick.`	Symbolic destination name for the target resource (1-8 characters).
`:luname.`	Identifies where the resource resides. (For our purposes, this is the virtual machine in which our partner program will execute.)
`:tpn.`	The transaction program name as it is known at the target LU.

We need to update either the SCOMDIR NAMES file or the UCOMDIR NAMES file to include a valid entry for our GETFILE symbolic destination name. The server program we are writing can be considered a private resource manager, so add an entry to the UCOMDIR NAMES file. If you do not have a UCOMDIR NAMES file, you can create one with the following entry (which is all you need in the file):

```
:nick.GETFILE  :luname.*USERID SERVR
               :tpn.GET
```

The GETFILE value for the `:nick.` tag corresponds to the *sym_dest_name* we have been specifying when we invoke our exec. For the `:luname.` tag, the *USERID is a keyword that indicates that our partner program is a private resource manager in the same TSAF collection, and SERVR identifies our partner's virtual machine (remember that if you used a different user ID for this virtual machine, you need to substitute that name here). GET, on the `:tpn.` tag, identifies the target private resource.

The order of the tags in the communications directory and the spacing between them will not make any difference, as long as all the tags for an entry are grouped together following the `:nick.` tag for that entry. If you put more than one tag on a line, separate them with at least one blank.

Save the additions to the UCOMDIR NAMES file, and try out the exec again. Enter

```
process getfile
```

and let the program default to the requested file name of TEST FILE A.

The resulting screen output should be:

```
process getfile Requesting the file:  TEST FILE A

Routine called:  CMINIT
  conversation_ID is 00000000
  sym_dest_name is GETFILE
  return_code is CM_OK

Routine called:  CMALLC
  conversation_ID is 00000000
  return_code is CM_OK

Routine called:  CMSEND

* ERROR:  An error occurred during a CMSEND call
          The return_code was set to CM_TPN_NOT_RECOGNIZED
Ready;
```

Figure 8. Output from PROCESS EXEC after Adding UCOMDIR NAMES Entry

We got the same error code of CM_TPN_NOT_RECOGNIZED again! The problem this time resulted from not entering a command called SET COMDIR after changing the UCOMDIR NAMES file. Basically, we did not let CMS know that we had updated a communications directory. This should give you an idea of the variety of reasons you can get this return code. It is hard to remember them without experiencing them a few times.

FYI: The SET COMDIR Command

The SET COMDIR command serves several functions. We are interested in just a couple of them right now. SET COMDIR FILE defines the names of both the system-level and the user-level communications directory files. The SET COMDIR ON BOTH command enables symbolic destination name resolution. The SYSPROF EXEC shipped with VM contains these statements, which are automatically issued when CMS IPLs the virtual machine.

When the SET COMDIR commands are executed, CMS makes an image of the two communications directories in memory. If you modify either of the communications directories, you need to enter the SET COMDIR RELOAD command so that a new image of the updated directories is made in memory.

For more information on the SET COMDIR command, see the *z/VM: CMS Commands and Utilities Reference*.

To find out what communications directory files are in memory, you can enter the QUERY COMDIR command.

If you modified an existing UCOMDIR NAMES file, enter:

```
set comdir reload
```

or if you had to create a UCOMDIR NAMES file, enter:

```
set comdir file user ucomdir names
```

and then execute the PROCESS EXEC again with

```
process getfile
```

Note: Now that we have an entry in the UCOMDIR NAMES file (and CMS knows about it), CPI Communications will be able to tell that GETFILE refers to a private resource manager. Until now, CPI Communications considered GETFILE to be a global resource, which is the default on VM when complete side information is not provided. Thus, when a global resource with the name GETFILE could not be located on the local system, it is assumed that the resource is elsewhere in the TSAF collection. The allocation request is then routed to the TSAF virtual machine on the local system. This helps to explain the CM_PRODUCT_SPECIFIC_ERROR that some users may have gotten in Step 2 the first time the program was executed with an Allocate call. Users working on a system that is not part of a TSAF collection received an error indicating that there was no TSAF virtual machine operating on their system. If you received that error, you can start executing the program again.

Depending on how your server virtual machine is set up, the output may not appear as we show it in this case. If it is not, do not be alarmed, just continue reading. Your screen output should be:

```
process getfile Requesting the file:  TEST FILE A

Routine called:  CMINIT
  conversation_ID is 00000000
  sym_dest_name is GETFILE
  return_code is CM_OK
```

Figure 9. Output from PROCESS EXEC after SET COMDIR Command

The program appears to wait following the Initialize_Conversation call. In fact, our application is hung because the partner virtual machine hasn't issued a SET SERVER ON command. The allocation request, therefore, cannot be presented to the server machine.

Note: If your results are different, it could be because there is already a SET SERVER ON command in the PROFILE EXEC of the server virtual machine. Read on through the next section to see if this seems to be the case. Also, you may see different results if there are active APPC/VM conversations in your virtual machine, for example, you may have an SFS directory accessed. See *FYI: SFS Directories Accessed* on page 29 for this case.

The SET SERVER ON command will enable interrupts and allow CMS private resource processing. Do not do anything from the REQUESTR user ID. Instead, from the SERVR user ID, enter

```
set server on
```

That command allows the allocation request to be presented, and the SERVR terminal should display the following message, with the appropriate time:

```
set server on
 hh:mm:ss  * MSG FROM SERVR  : DMSIUH2027E Connection request on path 0
 is severed for reason = 7
Ready;
```

Figure 10. System Response after Entering SET SERVER ON Command

More information is now displayed at the REQUESTR terminal, as well. The complete screen of information is:

```
process getfile Requesting the file:  TEST FILE A

Routine called:  CMINIT
  conversation_ID is 00000000
  sym_dest_name is GETFILE
  return_code is CM_OK

Routine called:  CMALLC
  conversation_ID is 00000000
  return_code is CM_OK

Routine called:  CMSEND

* ERROR:  An error occurred during a CMSEND call
          The return_code was set to CM_TPN_NOT_RECOGNIZED
Ready;
```

Figure 11. Output from PROCESS EXEC after Entering the SET SERVER ON Command from the SERVR Console

Although the *return_code* is CM_TPN_NOT_RECOGNIZED once again, it appears we are making progress because the output on the SERVR terminal indicates that some type of interaction has occurred between the requester and server virtual machines. But what caused the connection request to be severed, as noted in the message on the server side?

By looking up message DMSIUH2027E in the *z/VM: CMS and REXX/VM Messages and Codes* (or entering `help message dms2027e` on the command line to have the HELP Facility display the message), we can determine the meaning behind a reason code of 7 for the sever message we received. In general, a code of 7 indicates that resource or user ID validation has failed. The response suggested in the message description for code 7 mentions the $SERVER$ NAMES file.

“What’s a $SERVER$ NAMES file?” you might ask. It is another special CMS NAMES file that a server virtual machine uses to control access to the private resources it controls. Knowing this, you no doubt realize that the reason for the failure of our program is that we have not supplied an entry in the partner’s $SERVER$ NAMES file for the resource to which we want to connect.

Now seems like an appropriate time to focus our attention on the server application. We will start writing our server program and create a $SERVER$ NAMES entry to see if that helps the Send_Data (CMSEND) call to complete successfully.

FYI: SFS Directories Accessed

If your results are different from those on page 28, you may have active APPC/VM conversations in your virtual machine similar to the following.

```
  hh:mm:ss  * MSG FROM SERVR  : DMSIUH2027E Connection request on
  path 2 is severed for reason = 6
 Ready;
```

More information is now displayed at the REQUESTR terminal, as well. The complete screen of information is:

```
   process getfile Requesting the file:  TEST FILE A

Routine called:  CMINIT
  conversation_ID is 00000000
  sym_dest_name is GETFILE
  return_code is CM_OK

Routine called:  CMALLC
  conversation_ID is 00000000
  return_code is CM_OK

Routine called:  CMSEND

* ERROR:  An error occurred during a CMSEND call
          The return_code was set to CM_TPN_NOT_AVAILABLE_NO_RETRY
          Ready;
```

By looking up message DMSIUH2027E in the *z/VM: CMS and REXX/VM Messages and Codes* (or entering `help message dms2027e` on the command line to have the HELP Facility display the message), we can determine that the SET SERVER ON command needs to be issued.

Preparing the SERVR Virtual Machine

We need to create or modify four files on the SERVR virtual machine. One is the PROFILE EXEC. Another is the TEST FILE that's being requested. The third is the $SERVER$ NAMES file. And the fourth is the transaction program that corresponds to the name GET, which we provided in the UCOMDIR NAMES file of the REQUESTR virtual machine.

Modifying the PROFILE EXEC File

We need to add these three commands to the PROFILE EXEC to prepare the virtual machine to manage private resources:

```
SET SERVER ON
SET FULLSCREEN OFF
SET AUTOREAD OFF
```

The SET SERVER ON command enables CMS private resource processing. The SET FULLSCREEN OFF command ensures that CMS session services are deactivated. The SET AUTOREAD OFF command prevents CMS from issuing a console read immediately after command execution. This prevents the SERVR virtual machine from hanging when it gets autologged.

Complete the changes to the PROFILE EXEC, file it, and then run it by entering

```
profile
```

to put the new commands into effect.

Creating TEST FILE

Now we need to create the file we are requesting to have sent to us. Use TEST for the file name and FILE for the file type. Include a couple of lines of text in the file, such as:

```
This is the first line of the requested file.
This is the second line of the requested file.
```

Creating the $SERVER$ NAMES File

The program we will be writing soon for the server will be a private resource manager. VM controls access to a private resource through a special CMS NAMES file called $SERVER$ NAMES. This file contains the names of private resources and user IDs (virtual machines) that are allowed to connect to them.

Note: You can create or change your $SERVER$ NAMES file using the NAMES command with the SERVER option. See the NAMES command usage notes in the *z/VM: CMS Commands and Utilities Reference* for more information.

When an allocation request is received for a private resource, CMS checks the server virtual machine's $SERVER$ NAMES file for an entry that matches the private resource name specified as the target of the partner's allocation and determines if the requesting user ID is authorized to allocate to the private resource. If the user ID is authorized, CMS invokes the private resource with the resource name passed as a parameter. If the user ID is not authorized, the requester receives an allocation error of CM_SECURITY_NOT_VALID.

The $SERVER$ NAMES file has three tags:

Tag	What Value the Tag Specifies
`:nick.`	Specifies the name of the private resource (1-8 characters). This is the same value specified on the `:tpn.` tag in the requesting virtual machine's UCOMDIR (or SCOMDIR) NAMES file.
`:list.`	Specifies the users that are authorized for the private resource. This list can be individual user IDs, nicknames contained in a standard NAMES file that might refer to groups of users, or an ***** (asterisk) that specifies that any requester can connect to this private resource.
`:module.`	Specifies the name of the resource manager program for the private resource specified in the nickname field. This value is the name of a CMS module or exec that is to be invoked after connection authorization has been determined. If a value is not specified for the `:module.` tag, the value on the `:nick.` tag is used. So, the `:module.` tag can be omitted if the `:nick.` entry is identical to the CMS-invokable name of the private resource manager.

Let's create the $SERVER$ NAMES file and include the following information:

```
:nick.GET  :list.REQUESTR
           :module.SENDBACK
```

The GET value for the `:nick.` tag represents the name of the private resource. REQUESTR for the `:list.` tag value indicates that REQUESTR is the only user ID authorized to access this resource (if you used a different user ID for the REQUESTR virtual machine, be sure to specify the name you used here). The

:module. tag value SENDBACK is the name of the resource manager; this corresponds to the name of the transaction program that is to be invoked.

Note that the value specified for the :tpn. tag in UCOMDIR NAMES on the requester user ID is used as an index into the server's $SERVER$ NAMES file. For this reason, the values for both the :tpn. tag of UCOMDIR NAMES and the :nick. tag of $SERVER$ NAMES must match, as Figure 12 shows.

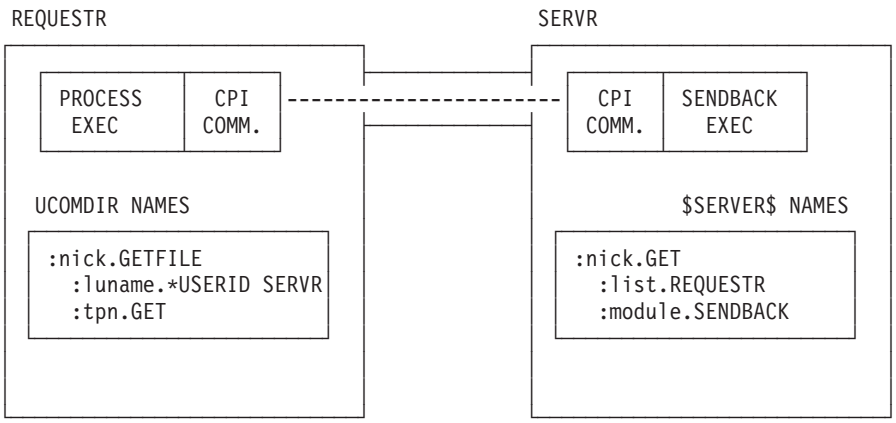


Figure 12. Relationship between UCOMDIR and $SERVER$ NAMES Files

Creating the SENDBACK EXEC File

The previous section indicated that the name of the PROCESS EXEC's partner program is SENDBACK. As was done in the requester application, begin SENDBACK EXEC with a REXX comment line (/* */) followed by the EXECIO routine to process and load the CMREXX COPY file.

The SENDBACK EXEC should initially contain the following lines (you can use the copy you made at the end of Step 1—or copy the PROCESS EXEC file now—and then change the first comment and remove the lines of code not listed here):

```
/*====================================================================*/
/*  SENDBACK EXEC - Sample server application.                        */
/*====================================================================*/

/*--------------------------------------------------------------------*/
/* Set up REXX environment for program-to-program communications      */
/* and enable error trapping of REXX errors.                          */
/*--------------------------------------------------------------------*/
address cpicomm
signal on error
/*--------------------------------------------------------------------*/
/* Equate pseudonyms to their integer values based on the             */
/* definitions contained in the CMREXX COPY file.                     */
/*--------------------------------------------------------------------*/
address command 'EXECIO * DISKR CMREXX COPY * (FINIS STEM PSEUDONYM.'
do index = 1 to pseudonym.0
  interpret pseudonym.index
end

GetOut:
  exit

/*------------------------- Subroutines --------------------------*/

TraceParms:
/*--------------------------------------------------------------------*/
```

```
/* Display parameters and their values as passed to this subroutine.*/
/*-------------------------------------------------------------------*/
parse arg parmlist
do word_num = 1 to words(parmlist)
  parameter = word(parmlist,word_num)
  select
    when (parameter = 'return_code') then
      say '  return_code is' cm_return_code.return_code
    otherwise
      say ' ' parameter 'is' value(parameter)
  end
end

return


Error:
/*-------------------------------------------------------------------*/
/* Report error when REXX special variable RC is not 0.             */
/*-------------------------------------------------------------------*/
say
say '* ERROR:  REXX has detected an error'
say '          The return code variable RC was set to' rc
signal GetOut


ErrorHandler:
/*-------------------------------------------------------------------*/
/* Report routine that failed and the error return code.            */
/*-------------------------------------------------------------------*/
parse arg routine_name
say
say '* ERROR:  An error occurred during a' routine_name 'call'
say '          The return_code was set to' cm_return_code.return_code
signal GetOut
```

Step 4. The Accept_Conversation (CMACCP) Call

The first CPI Communications routine we want to call from the SENDBACK EXEC is Accept_Conversation (CMACCP). The Accept_Conversation call is solely responsible for accepting incoming conversation requests. Like Initialize_Conversation (CMINIT), Accept_Conversation sets a number of conversation characteristics to default values (which we will discuss later) and assigns a conversation ID.

The values of the *conversation_type* and *sync_level* conversation characteristics are derived from the incoming allocation request and cannot be changed. Other conversation characteristics (such as *receive_type* and *send_type*) are set to their default values, but can be changed by Set calls anytime after issuing the Accept_Conversation call. The “Conversation Characteristics” section in the *Common Programming Interface Communications Reference* lists these characteristics and their default values.

Here is the format for Accept_Conversation:

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

Output Parameters

Both parameters on Accept_Conversation are for output. The ***conversation_ID*** parameter returns the conversation identifier assigned to the conversation by CPI Communications. This identifier will be used on all CPI Communications calls that follow on this conversation.

An additional point concerning the *conversation_ID* is that its value is not based on the value of its communications partner's *conversation_ID*. The *conversation_ID* only relates to the side of the conversation on which it was returned, either from an Initialize_Conversation call or an Accept_Conversation call. In other words, each side of the conversation gets its own conversation ID.

The values for ***return_code*** on an Accept_Conversation call that are of interest to us are:

CM_OK (0)
: indicates that the conversation has been accepted and the local program has entered **Receive** state.

CM_PRODUCT_SPECIFIC_ERROR (20)
: indicates that an error unique to the VM product has occurred. Check the CPICOMM LOGDATA file for a summary of the error.

CM_PROGRAM_STATE_CHECK (25)
: indicates that no incoming conversation exists.

Results of the Call

When *return_code* indicates CM_OK, the conversation enters **Receive** state and various conversation characteristics are initialized to their default values.

Adding CMACCP to Our Server Program

Now add the Accept_Conversation call to your SENDBACK EXEC. Again, we will want to examine the CPI Communications *return_code* parameter after each CPI Communications call as part of our minimal error checking. We will also want to display the results of the call execution.

The SENDBACK EXEC should now have the following lines in it:

```
/*====================================================================*/
/*  SENDBACK EXEC - Sample server application.                        */
/*====================================================================*/

/*--------------------------------------------------------------------*/
/* Set up REXX environment for program-to-program communications    */
/* and enable error trapping of REXX errors.                         */
/*--------------------------------------------------------------------*/
address cpicomm
signal on error
/*--------------------------------------------------------------------*/
/* Equate pseudonyms to their integer values based on the           */
/* definitions contained in the CMREXX COPY file.                    */
/*--------------------------------------------------------------------*/
address command 'EXECIO * DISKR CMREXX COPY * (FINIS STEM PSEUDONYM.'
do index = 1 to pseudonym.0
  interpret pseudonym.index
end
/*--------------------------------------------------------------------*/
/* Accept the incoming conversation.                                 */
/*--------------------------------------------------------------------*/
'CMACCP conversation_ID return_code'
say; say 'Routine called:  CMACCP'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMACCP'
call TraceParms 'conversation_ID return_code'
```

```
GetOut:
  exit

/*------------------------- Subroutines ---------------------------*/

  ⋮
```

Now file the exec, and execute it. From the SERVR virtual machine console, enter

```
sendback
```

Your results should be:

```
sendback

Routine called:  CMACCP

* ERROR:  An error occurred on a CMACCP call
          The return_code was set to CM_PROGRAM_STATE_CHECK
Ready;
```

Figure 13. Output Resulting from Execution of SENDBACK EXEC

Instead of CM_OK, the return code was CM_PROGRAM_STATE_CHECK. As we mentioned earlier, CM_PROGRAM_STATE_CHECK returned on an Accept_Conversation call means that no incoming conversation request was present. In simple terms, we did not start the PROCESS EXEC from the REQUESTR virtual machine, so there was no allocation request waiting to be accepted.

This time, run the PROCESS EXEC from the REQUESTR user ID, and you will see output at both terminals. After entering

```
process getfile
```

the results from the REQUESTR user ID should be:

```
process getfile
Requesting the file:  TEST FILE A

Routine called:  CMINIT
  conversation_ID is 00000000
  sym_dest_name is GETFILE
  return_code is CM_OK

Routine called:  CMALLC
  conversation_ID is 00000000
  return_code is CM_OK

Routine called:  CMSEND

* ERROR:  An error occurred during a CMSEND call
          The return_code was set to CM_RESOURCE_FAILURE_NO_RETRY
Ready;
```

Figure 14. Output from PROCESS EXEC Showing Step 4 Results

The results from the SERVR user ID should be:

```
Routine called:  CMACCP
  conversation_ID is 00000000
  return_code is CM_OK
Ready;
```

Figure 15. Output from SENDBACK EXEC Showing Step 4 Results

We have finally established a conversation between the two applications! CMS automatically started our server application based on information in the $SERVER$ NAMES file. But while the Accept_Conversation call appears to have completed successfully, there is a problem on the requester side with the Send_Data call. The *return_code* value of CM_RESOURCE_FAILURE_NO_RETRY arises because our server program terminates after it issues the Accept_Conversation call without first deallocating the conversation. Because the conversation was not explicitly deallocated, the termination is reported to the requester as an error indicating that the partner resource is no longer available.

We will be adding the Deallocate (CMDEAL) routine to our application a little later. But now, look at the next step for the server application in the starter set programs overview table 9, the Receive (CMRCV) call.

Step 5. The Receive (CMRCV) Call

The Receive (CMRCV) call receives information from an established conversation. The information received can be a data record, conversation status, or both.

The format for Receive is:

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

Input Parameters

Use the ***conversation_ID*** parameter to identify the conversation for which you want to issue this call. For SENDBACK EXEC, we will want to specify the value returned on the Accept_Conversation call.

For the ***requested_length*** parameter, we will specify the maximum amount of data that the program is prepared to receive with this Receive call. The range of valid *requested_length* values is from 0 to 32767. Remember, the amount of data that can be received by a single Receive call is limited by the value specified for the *requested_length* parameter.

Output Parameters

We use the ***buffer*** parameter to specify the variable that will hold the received data. The buffer will contain data only when *return_code* is set to CM_OK or CM_DEALLOCATED_NORMAL and *data_received* is set to a value indicating that at least some data was received. The buffer will not contain any data if *data_received* is set to CM_NO_DATA_RECEIVED.

The ***data_received*** parameter returns a value indicating whether the program received data. This parameter contains a value only when *return_code* is set to CM_OK or CM_DEALLOCATED_NORMAL. The possible *data_received* values that are of interest to us are:

CM_NO_DATA_RECEIVED (0)
: indicates that no data was received.

CM_COMPLETE_DATA_RECEIVED (2)
: indicates that a complete data record or the last remaining portion of the record was received.

CM_INCOMPLETE_DATA_RECEIVED (3)
: indicates that less than a complete data record was received. When the program receives CM_INCOMPLETE_DATA_RECEIVED for the *data_received* value, it should issue additional Receive (CMRCV) calls until an indication of CM_COMPLETE_DATA_RECEIVED is reported.

The ***received_length*** parameter returns the amount of data, in bytes, received by the program. The *received_length* parameter is not given a value when data is not received.

The ***status_received*** parameter returns a value that indicates the conversation status. It contains a value only when the *return_code* parameter is set to CM_OK. Valid values of interest to us for *status_received* are:

CM_NO_STATUS_RECEIVED (0)
: No status received; data may be present.

CM_SEND_RECEIVED (1)
: The remote program's end of the conversation has entered **Receive** state. The local program can now send data.

CM_CONFIRM_RECEIVED (2)
: The remote program has sent a confirmation request requesting the local program to respond by issuing a Confirmed call. The local program must respond by issuing Confirmed, Send_Error, or Deallocate with *deallocate_type* set to CM_DEALLOCATE_ABEND.

CM_CONFIRM_SEND_RECEIVED (3)
: The remote program's end of the conversation has entered **Receive** state with confirmation requested. The local program must respond by issuing Confirmed, Send_Error, or Deallocate with *deallocate_type* set to CM_DEALLOCATE_ABEND. Upon issuing a successful Confirmed call, the local program can send data.

The ***request_to_send_received*** parameter returns an indication of whether the remote program issued a Request_To_Send (CMRTS) call.

The ***return_code*** parameter values of interest to us now include:

CM_OK (0)
: indicates that the Receive call completed successfully.

CM_DEALLOCATED_ABEND (17)
: indicates that the remote program or the remote LU issued a Deallocate with *deallocate_type* set to CM_DEALLOCATE_ABEND. The conversation is now in **Reset** state.

CM_DEALLOCATED_NORMAL (18)
: indicates that the remote program issued a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL or CM_DEALLOCATE_FLUSH. If *deallocate_type* is CM_DEALLOCATE_SYNC_LEVEL, the *sync_level* is CM_NONE. The conversation is now in **Reset** state.

CM_PRODUCT_SPECIFIC_ERROR (20)
: indicates that an error unique to the VM product has occurred. Check the CPICOMM LOGDATA file for a summary of the error.

CM_PROGRAM_PARAMETER_CHECK (24)
: indicates that the specified conversation ID is unassigned or *requested_length* specifies a value greater than 32767.

CM_PROGRAM_STATE_CHECK (25)
: indicates that the conversation is not in an appropriate state to issue the Receive call.

CM_RESOURCE_FAILURE_NO_RETRY (26)
: indicates that a failure occurred that caused the conversation to be terminated prematurely or the remote program ended without deallocating the conversation. The VM appendix to the *Common Programming Interface Communications Reference* lists more specific possible explanations for this return code. The condition is not temporary. The conversation is in **Reset** state.

CM_RESOURCE_FAILURE_RETRY (27)
: indicates that a failure occurred that caused the conversation to be terminated prematurely. This could occur if the TSAF virtual machine encountered a problem during its processing or if the TSAF link went down. The condition may be temporary so the program can retry the allocation request. The conversation is in **Reset** state.

Results of the Call

For our purposes at this time, when *return_code* indicates CM_OK, the conversation enters or remains in **Receive** state. Other results are possible in various other scenarios; refer to the Receive call description in the *Common Programming Interface Communications Reference* for other results.

Adding a Receive (CMRCV) Loop to Our Server Program

Before calling Accept_Conversation, the conversation (from the server virtual machine's perspective) is in **Reset** state. Upon completion of that call, the conversation is in **Receive** state, so we can now receive the data that the PROCESS program is trying to send.

We will need to choose a value for the *requested_length* parameter. Because the only data that this program is going to receive is the name of the requested file, set the *requested_length* variable to 20, which should allow for the largest possible file name in VM.

The starter set programs table 9 indicates that we will be issuing the Receive calls from a loop in the server exec. We want to put the Receive call in a loop so our program will be able to avoid any dependency on a certain number of Receive calls. That way, it will be able to handle varying record lengths and receive data and status on either the same or separate Receive calls.

Both the Receive and Send_Data routines have a parameter called *buffer*, so our TraceParms subroutine needs to be able to distinguish between the two. Our program will display a received *buffer* whose length is returned by *received_length* and a *buffer* that was sent whose length is contained in *send_length*. So, we will use *receive_buffer* in place of *buffer* for the Receive call. But when TraceParms displays the parameter contents, we will still use *buffer* to label our output.

FYI: The LEFT Function in REXX

Before adding the Receive call to the SENDBACK EXEC, we will point out that when a REXX program calls Receive in VM, the *buffer* parameter upon return from Receive will have a size of 32767 bytes. Therefore, if the SAY instruction is used to display the contents of the *buffer*, the data received will be displayed along with pad characters for the remainder of the 32767 bytes.

To display only the data that was received on this call, we can take advantage of the REXX function LEFT, which will return the leftmost characters of a character string for a specified length. The best length to supply is the one returned on the Receive call itself, namely *received_length*.

Now add the Receive call and related instructions to your SENDBACK EXEC. The exec should have the following lines in it:

```
/*====================================================================*/
/*  SENDBACK EXEC - Sample server application.                        */
/*====================================================================*/

:
/*--------------------------------------------------------------------*/
/* Accept the incoming conversation.                                  */
/*--------------------------------------------------------------------*/
'CMACCP conversation_ID return_code'
say; say 'Routine called:  CMACCP'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMACCP'
call TraceParms 'conversation_ID return_code'
/*--------------------------------------------------------------------*/
/* Start a Receive loop.                                              */
/* Receive data, status, or both from conversation partner.           */
/*--------------------------------------------------------------------*/
requested_file = ''
requested_length = 20
do until (CMRCV_return_code ¬= CM_OK)
  'CMRCV conversation_ID receive_buffer requested_length',
        'data_received received_length status_received',
        'request_to_send_received return_code'
  CMRCV_return_code = return_code
  say; say 'Routine called:  CMRCV'
  select
    when (CMRCV_return_code = CM_OK) then
      do
        call TraceParms 'conversation_ID receive_buffer',
                        'requested_length data_received',
                        'received_length status_received',
                        'request_to_send_received return_code'
        if (data_received ¬= CM_NO_DATA_RECEIVED) then
          do
            receive_buffer = left(receive_buffer,received_length)
            requested_file = requested_file || receive_buffer
          end
      end
    otherwise
      call ErrorHandler 'CMRCV'
  end
end

GetOut:
  exit

/*------------------------- Subroutines --------------------------*/
```

```
TraceParms:
/*--------------------------------------------------------------------*/
/* Display parameters and their values as passed to this subroutine.*/
/*--------------------------------------------------------------------*/
parse arg parmlist
do word_num = 1 to words(parmlist)
  parameter = word(parmlist,word_num)
  select
    when (parameter = 'return_code') then
      say '  return_code is' cm_return_code.return_code
    when (parameter = 'receive_buffer') then
      say '  buffer is' left(receive_buffer,received_length)
    when (parameter = 'data_received') then
      say '  data_received is' cm_data_received.data_received
    when (parameter = 'status_received') then
      say '  status_received is' cm_status_received.status_received
    when (parameter = 'request_to_send_received') then
      say '  request_to_send_received is',
          cm_request_to_send_received.request_to_send_received
    otherwise
      say ' ' parameter 'is' value(parameter)
  end
end

return
```

⋮

File the exec and execute the PROCESS EXEC from the REQUESTR user ID by entering

```
process getfile
```

Your results from the REQUESTR user ID should be:

```
process getfile
Requesting the file:  TEST FILE A

Routine called:  CMINIT
  conversation_ID is 00000000
  sym_dest_name is GETFILE
  return_code is CM_OK

Routine called:  CMALLC
  conversation_ID is 00000000
  return_code is CM_OK

Routine called:  CMSEND
  conversation_ID is 00000000
  buffer is TEST FILE A
  send_length is 11
  request_to_send_received is CM_REQ_TO_SEND_NOT_RECEIVED
  return_code is CM_OK
Ready;
```

Figure 16. Output from PROCESS EXEC Showing Step 5 Results

Your results from the SERVR user ID should be:

```
Routine called:  CMACCP
  conversation_ID is 00000000
  return_code is CM_OK

Routine called:  CMRCV
  conversation_ID is 00000000
  buffer is TEST FILE A
  requested_length is 20
  data_received is CM_COMPLETE_DATA_RECEIVED
  received_length is 11
  status_received is CM_NO_STATUS_RECEIVED
  request_to_send_received is CM_REQ_TO_SEND_NOT_RECEIVED
  return_code is CM_OK

Routine called:  CMRCV

* ERROR:  An error occurred during a CMRCV call
          The return_code was set to CM_RESOURCE_FAILURE_NO_RETRY
Ready;
```

Figure 17. Output from SENDBACK EXEC Showing Step 5 Results

The final *return_code* value of CM_RESOURCE_FAILURE_NO_RETRY on the SERVR user ID indicates that the partner program terminated abnormally. We will continue to ignore this error for the time being.

A couple of other output parameters from the server program's Receive call are of interest. The server program successfully received the name of the requested file in *buffer*. The *data_received* value of CM_COMPLETE_DATA_RECEIVED informs us that we received a complete data record. And the values for *status_received* and *request_to_send_received* indicate that neither status nor a request-to-send notification was received on the call.

Before the contents of the requested file can be sent to the partner program, the direction of the conversation needs to be reversed so that the SENDBACK EXEC in the SERVR virtual machine is in **Send** state. Because the partner in **Send** state controls the conversation, only it can reverse the direction. We will also need to prepare the PROCESS EXEC to receive the TEST FILE data.

Step 6. Adding a Receive (CMRCV) Loop to Our Requester Program

You can also use Receive to change the conversation state from **Send** to **Receive** state. When the Receive call is issued from **Send** state with *receive_type* set to its default value of CM_RECEIVE_AND_WAIT, the local system sends any buffered information to the remote program. The partner is notified by receipt of a *status_received* value of CM_SEND_RECEIVED that it may begin sending data. The local conversation's switch into **Receive** state occurs when the Receive call completes.

Switching states in this manner will be useful to us because the PROCESS EXEC will have to call Receive anyway to get the information that the server will be sending to it.

The number and length of data lines in TEST FILE is not known to the requester, so there may be no way to determine how much data will be available to be received. It is important to remember that the length of data records in the context of communications programs does not generally coincide with the logical record length of the lines or records of a file. The records (lines) in the file whose contents our

program will be sending to another user may be 80 bytes long, but the data records used by the communications program sending the data may be 2KB while the data records received at the other end of the conversation may be 4KB long. These length values are quite arbitrary and are based on the requirements for a particular application.

Because our programs are very simple and we want to demonstrate how the various CPI Communications calls work together, we will establish the convention of sending and receiving 80-byte data records. In our particular case, these records will coincide with the records of TEST FILE.

The PROCESS EXEC could receive the data by issuing multiple Receive calls. By calling Receive from within a loop, however, the application can handle this situation by coding only a single call to Receive. The first time through the loop, the Receive call will change the state and, when it becomes available, receive the first 80-byte data record sent by the server. Subsequent passes through the loop will receive the rest of the records of the file.

Using a record size of 80 bytes will not be the best approach for all applications. In this case, each Receive will return one complete line of the requested file, but the overhead of multiple Receive calls will not be appropriate for some situations. Setting the send and receive lengths to larger values would overcome this potential drawback.

The starter set programs table 9 shows that the requester application will deallocate the conversation after it has finished receiving data from the server. Because we are adding Receive to the PROCESS EXEC inside a loop, we will need to decide when the looping should be terminated so the Deallocate call can be made.

After sending all the contents of TEST FILE to the requester, the server will switch its end of the conversation back to **Receive** state. The requester will be notified of this change through the *status_received* parameter, which will be set to CM_SEND_RECEIVED. The receipt of this status, then, will be a signal for the requester to end the Receive loop.

Within the loop, the requester will need to process the incoming data. Let's use EXECIO to add each data record to a file called OUTPUT LOGFILE. Of course, you could choose any name you like.

Here are the contents of PROCESS EXEC at this point:

```
/*================================================================*/
/*  PROCESS EXEC - Sample file requester application.             */
/*================================================================*/

⋮
/*----------------------------------------------------------------*/
/* Send the name of the file being requested to the partner program.*/
/*----------------------------------------------------------------*/
buffer = fname ftype fmode
send_length = length(buffer)
'CMSEND conversation_ID buffer send_length',
       'request_to_send_received return_code'
say; say 'Routine called:  CMSEND'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMSEND'
call TraceParms 'conversation_ID buffer send_length',
                'request_to_send_received return_code'
/*----------------------------------------------------------------*/
/* Start a Receive loop.  Receive calls will be issued until      */
/* notification that the partner has finished sending data and    */
```

```
/* entered Receive state at its end of the conversation (noted by   */
/* receipt of CM_SEND_RECEIVED                                      */
/*------------------------------------------------------------------*/
complete_line = ''
requested_length = 80
do until (status_received = CM_SEND_RECEIVED)
  /*----------------------------------------------------------------*/
  /* Receive information from the conversation partner.             */
  /*----------------------------------------------------------------*/
  'CMRCV conversation_ID receive_buffer requested_length',
        'data_received received_length status_received',
        'request_to_send_received return_code'
  say; say 'Routine called:  CMRCV'
  select
    when (return_code = CM_OK) then
      do
        call TraceParms 'conversation_ID receive_buffer',
                        'requested_length data_received',
                        'received_length status_received',
                        'request_to_send_received return_code'
        if (data_received ¬= CM_NO_DATA_RECEIVED) then
          do
            receive_buffer = left(receive_buffer,received_length)
            complete_line = complete_line || receive_buffer
          end
        if (data_received = CM_COMPLETE_DATA_RECEIVED) then
          do
            /*--------------------------------------------------*/
            /* Use EXECIO to write the data to OUTPUT LOGFILE A     */
            /* and reset the complete_line variable to nulls.       */
            /*--------------------------------------------------*/
            address command 'EXECIO 1 DISKW OUTPUT LOGFILE A (FINIS',
                            'STRING' complete_line
            complete_line = ''
          end
      end
    otherwise
      call ErrorHandler 'CMRCV'
  end
end

GetOut:
  exit

/*------------------------- Subroutines --------------------------*/

TraceParms:
/*------------------------------------------------------------------*/
/* Display parameters and their values as passed to this subroutine.*/
/*------------------------------------------------------------------*/
parse arg parmlist
do word_num = 1 to words(parmlist)
  parameter = word(parmlist,word_num)
  select
    when (parameter = 'return_code') then
      say '  return_code is' cm_return_code.return_code
    when (parameter = 'buffer') then
      say '  buffer is' left(buffer,send_length)
    when (parameter = 'receive_buffer') then
      say '  buffer is' left(receive_buffer,received_length)
    when (parameter = 'data_received') then
      say '  data_received is' cm_data_received.data_received
    when (parameter = 'status_received') then
      say '  status_received is' cm_status_received.status_received
    when (parameter = 'request_to_send_received') then
      say '  request_to_send_received is',
          cm_request_to_send_received.request_to_send_received
```

```
    otherwise
      say ' ' parameter 'is' value(parameter)
  end
end

return
```

⋮

File the exec, and try out our changes. From a command line at the REQUESTR virtual machine, enter

```
process getfile
```

The REQUESTR side results should be:

```
process getfile
Requesting the file:  TEST FILE A

Routine called:  CMINIT
  conversation_ID is 00000000
  sym_dest_name is GETFILE
  return_code is CM_OK

Routine called:  CMALLC
  conversation_ID is 00000000
  return_code is CM_OK

Routine called:  CMSEND
  conversation_ID is 00000000
  buffer is TEST FILE A
  send_length is 11
  request_to_send_received is CM_REQ_TO_SEND_NOT_RECEIVED
  return_code is CM_OK

Routine called:  CMRCV
  conversation_ID is 00000000
  buffer is
  requested_length is 80
  data_received is CM_NO_DATA_RECEIVED
  received_length is 0
  status_received is CM_SEND_RECEIVED
  request_to_send_received is CM_REQ_TO_SEND_NOT_RECEIVED
  return_code is CM_OK
Ready;
```

Figure 18. Output from PROCESS EXEC Showing Step 6 Results

The results from the SERVR user ID should be:

```
Routine called:  CMACCP
  conversation_ID is 00000000
  return_code is CM_OK

Routine called:  CMRCV
  conversation_ID is 00000000
  buffer is TEST FILE A
  requested_length is 20
  data_received is CM_COMPLETE_DATA_RECEIVED
  received_length is 11
  status_received is CM_NO_STATUS_RECEIVED
  request_to_send_received is CM_REQ_TO_SEND_NOT_RECEIVED
  return_code is CM_OK

Routine called:  CMRCV
  conversation_ID is 00000000
  buffer is
  requested_length is 20
  data_received is CM_NO_DATA_RECEIVED
  received_length is 0
  status_received is CM_SEND_RECEIVED
  request_to_send_received is CM_REQ_TO_SEND_NOT_RECEIVED
  return_code is CM_OK

Routine called:  CMRCV

* ERROR:  An error occurred during a CMRCV call
          The return_code was set to CM_RESOURCE_FAILURE_NO_RETRY
Ready;
```

Figure 19. Output from SENDBACK EXEC Showing Step 6 Results

Several new interactions that are worth examining occurred between the programs.

After PROCESS EXEC sent the name of the desired file to the partner, it entered the new Receive loop. The Receive call issued by the requester from **Send** state sent a notification that its partner could begin to send data. The server program was presented with that send notification on its second Receive call, which was indicated by the *status_received* value of CM_SEND_RECEIVED.

Now, here is where it gets interesting. You will have to trust us on this for the moment, but when the server program was presented with the *status_received* value of CM_SEND_RECEIVED, the server's side of the conversation entered **Send** state. Because the server program is also executing a Receive loop, it issued another Receive call. Because the server issued this Receive call from **Send** state, send control for the conversation was passed back to the requester! (Yes, the conversation's direction was switched again before any data could be sent by the server.)

At this point, the requester's Receive completed. And you will note that the *status_received* parameter was set to CM_SEND_RECEIVED, which is the condition that completes the requester's Receive loop. The requester's program then ends, which is reflected to the server program by the *return_code* of CM_RESOURCE_FAILURE_NO_RETRY.

Now that the server is receiving the change-of-direction notification, we are ready to update the SENDBACK EXEC to send the requested file contents to the requester. (And we will need to keep the server in **Send** state long enough to accomplish that task.)

Step 7. Adding a Send_Data (CMSEND) Loop to Our Server

We will set up a Send_Data loop in the SENDBACK EXEC similar to the Receive loop in the PROCESS EXEC. As we mentioned in the discussion 42, we will assume 80-byte data records. Again, this approach may not be the best approach for all applications, because the overhead of multiple Send_Data calls may not be appropriate. It may be far more practical to use data records large enough to send an entire file at once.

The requester exec expects the server to turn the conversation around when it finishes sending the contents of the file, so issue a Receive (CMRCV) call following the Send_Data loop. Issuing the Receive will place the server's end of the conversation in **Receive** state and notify the requester that it has entered **Send** state again.

It seems that calling a Send subroutine from within the Receive loop will handle our situation. The only time that the server program will get send control for the conversation is when the requester program is ready to receive the file contents. So, a *status_received* value of CM_SEND_RECEIVED will be the indication for our server application that it is time to send the file.

The SENDBACK EXEC should now have the following lines in it:

```
/*====================================================================*/
/*  SENDBACK EXEC - Sample server application.                        */
/*====================================================================*/

:
/*--------------------------------------------------------------------*/
/* Start a Receive loop.                                              */
/* Receive data, status, or both from conversation partner.           */
/*--------------------------------------------------------------------*/
requested_file = ''
requested_length = 20
do until (CMRCV_return_code ¬= CM_OK)
  'CMRCV conversation_ID receive_buffer requested_length',
        'data_received received_length status_received',
        'request_to_send_received return_code'
  CMRCV_return_code = return_code
  say; say 'Routine called:  CMRCV'
  select
    when (CMRCV_return_code = CM_OK) then
      do
        call TraceParms 'conversation_ID receive_buffer',
                        'requested_length data_received',
                        'received_length status_received',
                        'request_to_send_received return_code'
        if (data_received ¬= CM_NO_DATA_RECEIVED) then
          do
            receive_buffer = left(receive_buffer,received_length)
            requested_file = requested_file || receive_buffer
          end
        if (status_received = CM_SEND_RECEIVED) then
          call SendFile
      end
    otherwise
      call ErrorHandler 'CMRCV'
  end
end

GetOut:
  exit

/*------------------------- Subroutines --------------------------*/
```

```
SendFile:
/*-------------------------------------------------------------------*/
/* Read the contents of the requested file and send each line of     */
/* the file to the partner program.                                  */
/*-------------------------------------------------------------------*/
address command 'EXECIO * DISKR' requested_file '(FINIS STEM LINE.'
do index = 1 to line.0
  buffer = line.index
  send_length = length(buffer)
  'CMSEND conversation_ID buffer send_length',
         'request_to_send_received return_code'
  say; say 'Routine called:  CMSEND'
  if (return_code ¬= CM_OK) then call ErrorHandler 'CMSEND'
  call TraceParms 'conversation_ID buffer send_length',
                  'request_to_send_received return_code'
end

return


TraceParms:
/*-------------------------------------------------------------------*/
/* Display parameters and their values as passed to this subroutine.*/
/*-------------------------------------------------------------------*/
parse arg parmlist
do word_num = 1 to words(parmlist)
  parameter = word(parmlist,word_num)
  select
    when (parameter = 'return_code') then
      say '  return_code is' cm_return_code.return_code
    when (parameter = 'buffer') then
      say '  buffer is' left(buffer,send_length)
    when (parameter = 'receive_buffer') then
      say '  buffer is' left(receive_buffer,received_length)
    when (parameter = 'data_received') then
      say '  data_received is' cm_data_received.data_received
    when (parameter = 'status_received') then
      say '  status_received is' cm_status_received.status_received
    when (parameter = 'request_to_send_received') then
      say '  request_to_send_received is',
          cm_request_to_send_received.request_to_send_received
    otherwise
      say ' ' parameter 'is' value(parameter)
  end
end
return
```

⋮

File the SENDBACK EXEC. Now that both execs have been updated, enter

```
process getfile
```

from the REQUESTR user ID. PROCESS EXEC will keep issuing the Receive call from the loop until all the data has been received. As each line of TEST FILE is received into the requester's *buffer* parameter, it will be displayed.

The results on the REQUESTR virtual machine should be:

```
process getfile
Requesting the file:  TEST FILE A

Routine called:  CMINIT
  conversation_ID is 00000000
  sym_dest_name is GETFILE
  return_code is CM_OK

Routine called:  CMALLC
  conversation_ID is 00000000
  return_code is CM_OK

Routine called:  CMSEND
  conversation_ID is 00000000
  buffer is TEST FILE A
  send_length is 11
  request_to_send_received is CM_REQ_TO_SEND_NOT_RECEIVED
  return_code is CM_OK

Routine called:  CMRCV
  conversation_ID is 00000000
  buffer is This is the first line of the requested file.

  requested_length is 80
  data_received is CM_COMPLETE_DATA_RECEIVED
  received_length is 80
  status_received is CM_NO_STATUS_RECEIVED
  request_to_send_received is CM_REQ_TO_SEND_NOT_RECEIVED
  return_code is CM_OK

Routine called:  CMRCV
  conversation_ID is 00000000
  buffer is This is the second line of the requested file.

  requested_length is 80
  data_received is CM_COMPLETE_DATA_RECEIVED
  received_length is 80
  status_received is CM_NO_STATUS_RECEIVED
  request_to_send_received is CM_REQ_TO_SEND_NOT_RECEIVED
  return_code is CM_OK

Routine called:  CMRCV
  conversation_ID is 00000000
  buffer is
  requested_length is 80
  data_received is CM_NO_DATA_RECEIVED
  received_length is 0
  status_received is CM_SEND_RECEIVED
  request_to_send_received is CM_REQ_TO_SEND_NOT_RECEIVED
  return_code is CM_OK
Ready;
```

Figure 20. Output from PROCESS EXEC Showing Step 7 Results

Congratulations! You have just successfully requested and received a small file's worth of information from a file server using the starter set of SAA CPI Communications routines.

The lines displayed for CMRCV could be repeated a number of times depending on the size of the TEST FILE you created. We have written our program to assume that when the *status_received* value is CM_SEND_RECEIVED, all of the file contents have been received. (In reality, however, this value indicates only that this end of the conversation is back in **Send** state.) Also at this point, the contents of TEST FILE have been written to the OUTPUT LOGFILE on the REQUESTR user

ID. Note that our application does not preserve certain file characteristics of the original file, such as logical record length (LRECL) and record format (RECFM), when it creates the new file on the REQUESTR user ID.

This time, the final routine call from PROCESS EXEC produces a *return_code* of CM_OK because the requester is the first of the two execs to terminate. The server exec, however, is waiting for information from its conversation partner, so we would expect the final routine in SENDBACK EXEC to complete with the familiar *return_code* value of CM_RESOURCE_FAILURE_NO_RETRY. We should not see that return code after we add the Deallocate (CMDEAL) call, but let us see if it shows up this time as we are expecting.

The results from the SERVR user ID should be:

```
Routine called:  CMACCP
  conversation_ID is 00000000
  return_code is CM_OK

Routine called:  CMRCV
  conversation_ID is 00000000
  buffer is TEST FILE A
  requested_length is 20
  data_received is CM_COMPLETE_DATA_RECEIVED
  received_length is 11
  status_received is CM_NO_STATUS_RECEIVED
  request_to_send_received is CM_REQ_TO_SEND_NOT_RECEIVED
  return_code is CM_OK

Routine called:  CMRCV
  conversation_ID is 00000000
  buffer is
  requested_length is 20
  data_received is CM_NO_DATA_RECEIVED
  received_length is 0
  status_received is CM_SEND_RECEIVED
  request_to_send_received is CM_REQ_TO_SEND_NOT_RECEIVED
  return_code is CM_OK

Routine called:  CMSEND
  conversation_ID is 00000000
  buffer is This is the first line of the requested file.

  send_length is 80
  request_to_send_received is CM_REQ_TO_SEND_NOT_RECEIVED
  return_code is CM_OK

Routine called:  CMSEND
  conversation_ID is 00000000
  buffer is This is the second line of the requested file.

  send_length is 80
  request_to_send_received is CM_REQ_TO_SEND_NOT_RECEIVED
  return_code is CM_OK

Routine called:  CMRCV

* ERROR:  An error occurred during a CMRCV call
          The return_code was set to CM_RESOURCE_FAILURE_NO_RETRY
Ready;
```

Figure 21. Output from SENDBACK EXEC Showing Step 7 Results

Not surprisingly, the final *return_code* value was set to CM_RESOURCE_FAILURE_NO_RETRY.

Now, according to our starter set programs table 9, there is only one step left, and that is to deallocate the conversation from the requester exec. This step requires adding a Deallocate call to the PROCESS EXEC. When SENDBACK has finished sending the contents of TEST FILE, control returns to its Receive loop. A Receive call is then issued from **Send** state to reverse the direction of the conversation and receive the deallocation notification.

Step 8. The Deallocate (CMDEAL) Call

The Deallocate (CMDEAL) call ends a conversation. When the Deallocate call completes successfully, the *conversation_ID* is no longer assigned.

The format for Deallocate is:

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

Input Parameter

Use the ***conversation_ID*** parameter to identify the conversation.

Output Parameter

The ***return_code*** is dependent on the *deallocate_type* and *sync_level* conversation characteristics. Using the default values of CM_DEALLOCATE_SYNC_LEVEL and CM_NONE, respectively, for those characteristics, the possible values for the *return_code* variable are:

CM_OK (0)
: indicates that the conversation deallocation completed successfully.

CM_PRODUCT_SPECIFIC_ERROR (20)
: indicates that an error unique to the VM product has occurred. Check the CPICOMM LOGDATA file for a summary of the error.

CM_PROGRAM_PARAMETER_CHECK (24)
: indicates that the specified conversation ID is unassigned.

CM_PROGRAM_STATE_CHECK (25)
: indicates that the conversation is not in **Send** or **Send-Pending** state.

Results of the Call

After the Deallocate call completes successfully, the conversation is considered to have entered **Reset** state, basically meaning that there is nothing left of the conversation.

Adding CMDEAL to Our Requester Program

Let's add the Deallocate (CMDEAL) call to the PROCESS EXEC. The complete exec should now have the following lines in it:

```
/*====================================================================*/
/*  PROCESS EXEC - Sample file requester application.                 */
/*====================================================================*/

arg sym_dest_name fname ftype fmode .           /* get user's input */
/*--------------------------------------------------------------------*/
/* If a file was not specifically requested, set up a default.        */
/*--------------------------------------------------------------------*/
if (fname = '') then
  do
    fname = 'TEST'
    ftype = 'FILE'
```

```
    fmode = 'A'
  end
say 'Requesting the file: ' fname ftype fmode
/*-------------------------------------------------------------------*/
/* Set up REXX environment for program-to-program communications    */
/* and enable error trapping of REXX errors.                         */
/*-------------------------------------------------------------------*/
address cpicomm
signal on error
/*-------------------------------------------------------------------*/
/* Equate pseudonyms to their integer values based on the           */
/* definitions contained in the CMREXX COPY file.                    */
/*-------------------------------------------------------------------*/
address command 'EXECIO * DISKR CMREXX COPY * (FINIS STEM PSEUDONYM.'
do index = 1 to pseudonym.0
  interpret pseudonym.index
end
/*-------------------------------------------------------------------*/
/* Initialize the conversation.                                      */
/*-------------------------------------------------------------------*/
'CMINIT conversation_ID sym_dest_name return_code'
say; say 'Routine called:  CMINIT'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMINIT'
call TraceParms 'conversation_ID sym_dest_name return_code'
/*-------------------------------------------------------------------*/
/* Allocate the conversation.                                        */
/*-------------------------------------------------------------------*/
'CMALLC conversation_ID return_code'
say; say 'Routine called:  CMALLC'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMALLC'
call TraceParms 'conversation_ID return_code'
/*-------------------------------------------------------------------*/
/* Send the name of the file being requested to the partner program.*/
/*-------------------------------------------------------------------*/
buffer = fname ftype fmode
send_length = length(buffer)
'CMSEND conversation_ID buffer send_length',
       'request_to_send_received return_code'
say; say 'Routine called:  CMSEND'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMSEND'
call TraceParms 'conversation_ID buffer send_length',
                'request_to_send_received return_code'
/*-------------------------------------------------------------------*/
/* Start a Receive loop.  Receive calls will be issued until        */
/* notification that the partner has finished sending data and      */
/* entered Receive state at its end of the conversation (noted by   */
/* receipt of CM_SEND_RECEIVED                                       */
/* for status_received) or until a return_code value other than     */
/* CM_OK is returned.  The record length of the incoming data       */
/* is assumed to be 80 bytes, or less.                               */
/*-------------------------------------------------------------------*/
complete_line = ''
requested_length = 80
do until (status_received = CM_SEND_RECEIVED)
  /*-----------------------------------------------------------------*/
  /* Receive information from the conversation partner.             */
  /*-----------------------------------------------------------------*/
  'CMRCV conversation_ID receive_buffer requested_length',
        'data_received received_length status_received',
        'request_to_send_received return_code'
  say; say 'Routine called:  CMRCV'
  select
    when (return_code = CM_OK) then
      do
        call TraceParms 'conversation_ID receive_buffer',
                        'requested_length data_received',
                        'received_length status_received',
```

```
                    'request_to_send_received return_code'
        if (data_received ¬= CM_NO_DATA_RECEIVED) then
          do
            receive_buffer = left(receive_buffer,received_length)
            complete_line = complete_line || receive_buffer
          end
        if (data_received = CM_COMPLETE_DATA_RECEIVED) then
          do
            /*-------------------------------------------------------*/
            /* Use EXECIO to write the data to OUTPUT LOGFILE A      */
            /* and reset the complete_line variable to nulls.        */
            /*-------------------------------------------------------*/
            address command 'EXECIO 1 DISKW OUTPUT LOGFILE A (FINIS',
                            'STRING' complete_line
            complete_line = ''
          end
      end
    otherwise
      call ErrorHandler 'CMRCV'
  end
end
/*---------------------------------------------------------------------*/
/* Deallocate the conversation normally.                               */
/*---------------------------------------------------------------------*/
'CMDEAL conversation_ID return_code'
say; say 'Routine called:  CMDEAL'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMDEAL'
call TraceParms 'conversation_ID return_code'

GetOut:
  exit

/*-------------------------- Subroutines --------------------------*/

TraceParms:
/*---------------------------------------------------------------------*/
/* Display parameters and their values as passed to this subroutine.*/
/*---------------------------------------------------------------------*/
parse arg parmlist
do word_num = 1 to words(parmlist)
  parameter = word(parmlist,word_num)
  select
    when (parameter = 'return_code') then
      say '  return_code is' cm_return_code.return_code
    when (parameter = 'buffer') then
      say '  buffer is' left(buffer,send_length)
    when (parameter = 'receive_buffer') then
      say '  buffer is' left(receive_buffer,received_length)
    when (parameter = 'data_received') then
      say '  data_received is' cm_data_received.data_received
    when (parameter = 'status_received') then
      say '  status_received is' cm_status_received.status_received
    when (parameter = 'request_to_send_received') then
      say '  request_to_send_received is',
          cm_request_to_send_received.request_to_send_received
    otherwise
      say ' ' parameter 'is' value(parameter)
  end
end

return


Error:
/*---------------------------------------------------------------------*/
/* Report error when REXX special variable RC is not 0.                */
/*---------------------------------------------------------------------*/
```

```
say
say '* ERROR:  REXX has detected an error'
say '          The return code variable RC was set to' rc
signal GetOut


ErrorHandler:
/*--------------------------------------------------------------------*/
/* Report routine that failed and the error return code.              */
/*--------------------------------------------------------------------*/
parse arg routine_name
say
say '* ERROR:  An error occurred during a' routine_name 'call'
say '          The return_code was set to' cm_return_code.return_code
signal GetOut
```

File the exec and, once more, enter

```
process getfile
```

from the REQUESTR user ID command line.

The results on the REQUESTR virtual machine should be:

```
process getfile
Requesting the file:  TEST FILE A

Routine called:  CMINIT
  conversation_ID is 00000000
  sym_dest_name is GETFILE
  return_code is CM_OK

Routine called:  CMALLC
  conversation_ID is 00000000
  return_code is CM_OK

Routine called:  CMSEND
  conversation_ID is 00000000
  buffer is TEST FILE A
  send_length is 11
  request_to_send_received is CM_REQ_TO_SEND_NOT_RECEIVED
  return_code is CM_OK

Routine called:  CMRCV
  conversation_ID is 00000000
  buffer is This is the first line of the requested file.

  requested_length is 80
  data_received is CM_COMPLETE_DATA_RECEIVED
  received_length is 80
  status_received is CM_NO_STATUS_RECEIVED
  request_to_send_received is CM_REQ_TO_SEND_NOT_RECEIVED
  return_code is CM_OK

Routine called:  CMRCV
  conversation_ID is 00000000
  buffer is This is the second line of the requested file.

  requested_length is 80
  data_received is CM_COMPLETE_DATA_RECEIVED
  received_length is 80
  status_received is CM_NO_STATUS_RECEIVED
  request_to_send_received is CM_REQ_TO_SEND_NOT_RECEIVED
  return_code is CM_OK

Routine called:  CMRCV
  conversation_ID is 00000000
  buffer is
  requested_length is 80
  data_received is CM_NO_DATA_RECEIVED
  received_length is 0
  status_received is CM_SEND_RECEIVED
  request_to_send_received is CM_REQ_TO_SEND_NOT_RECEIVED
  return_code is CM_OK

Routine called:  CMDEAL
  conversation_ID is 00000000
  return_code is CM_OK
Ready;
```

Figure 22. Output from PROCESS EXEC Showing Step 8 Results

The results on the SERVR side of the conversation should be:

```
Routine called:  CMACCP
  conversation_ID is 00000000
  return_code is CM_OK

Routine called:  CMRCV
  conversation_ID is 00000000
  buffer is TEST FILE A
  requested_length is 20
  data_received is CM_COMPLETE_DATA_RECEIVED
  received_length is 11
  status_received is CM_NO_STATUS_RECEIVED
  request_to_send_received is CM_REQ_TO_SEND_NOT_RECEIVED
  return_code is CM_OK

Routine called:  CMRCV
  conversation_ID is 00000000
  buffer is
  requested_length is 20
  data_received is CM_NO_DATA_RECEIVED
  received_length is 0
  status_received is CM_SEND_RECEIVED
  request_to_send_received is CM_REQ_TO_SEND_NOT_RECEIVED
  return_code is CM_OK

Routine called:  CMSEND
  conversation_ID is 00000000
  buffer is This is the first line of the requested file.

  send_length is 80
  request_to_send_received is CM_REQ_TO_SEND_NOT_RECEIVED
  return_code is CM_OK

Routine called:  CMSEND
  conversation_ID is 00000000
  buffer is This is the second line of the requested file.

  send_length is 80
  request_to_send_received is CM_REQ_TO_SEND_NOT_RECEIVED
  return_code is CM_OK

Routine called:  CMRCV

* ERROR:  An error occurred during a CMRCV call
          The return_code was set to CM_DEALLOCATED_NORMAL
Ready;
```

Figure 23. Output from SENDBACK EXEC Showing Step 8 Results

The *return_code* value of CM_DEALLOCATED_NORMAL indicates to the server that its partner deallocated the conversation. The conversation now has entered **Reset** state on the server's end as well.

Although the *return_code* is not set to CM_OK, a value of CM_DEALLOCATED_NORMAL does not reflect an error condition. Rather, it is the indication of a normal termination.

Let's quickly update the SENDBACK EXEC so we will not flag this condition as an error.

Here is the complete updated exec:

```
/*====================================================================*/
/*  SENDBACK EXEC - Sample server application.                        */
/*====================================================================*/

/*--------------------------------------------------------------------*/
/* Set up REXX environment for program-to-program communications    */
/* and enable error trapping of REXX errors.                         */
/*--------------------------------------------------------------------*/
address cpicomm
signal on error
/*--------------------------------------------------------------------*/
/* Equate pseudonyms to their integer values based on the           */
/* definitions contained in the CMREXX COPY file.                    */
/*--------------------------------------------------------------------*/
address command 'EXECIO * DISKR CMREXX COPY * (FINIS STEM PSEUDONYM.'
do index = 1 to pseudonym.0
  interpret pseudonym.index
end
/*--------------------------------------------------------------------*/
/* Accept the incoming conversation.                                 */
/*--------------------------------------------------------------------*/
'CMACCP conversation_ID return_code'
say; say 'Routine called:  CMACCP'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMACCP'
call TraceParms 'conversation_ID return_code'
/*--------------------------------------------------------------------*/
/* Start a Receive loop.                                             */
/* Receive data, status, or both from conversation partner.         */
/*--------------------------------------------------------------------*/
requested_file = ''
requested_length = 20
do until (CMRCV_return_code ¬= CM_OK)
  'CMRCV conversation_ID receive_buffer requested_length',
        'data_received received_length status_received',
        'request_to_send_received return_code'
  CMRCV_return_code = return_code
  say; say 'Routine called:  CMRCV'
  select
    when (CMRCV_return_code = CM_OK) then
      do
        call TraceParms 'conversation_ID receive_buffer',
                        'requested_length data_received',
                        'received_length status_received',
                        'request_to_send_received return_code'
        if (data_received ¬= CM_NO_DATA_RECEIVED) then
          do
            receive_buffer = left(receive_buffer,received_length)
            requested_file = requested_file || receive_buffer
          end
        if (status_received = CM_SEND_RECEIVED) then
          call SendFile
      end
    when (CMRCV_return_code = CM_DEALLOCATED_NORMAL) then
      do
        call TraceParms 'conversation_ID receive_buffer',
                        'requested_length data_received',
                        'received_length status_received',
                        'request_to_send_received return_code'
        say; say 'Conversation deallocated by partner'
      end
    otherwise
      call ErrorHandler 'CMRCV'
  end
end

GetOut:
  exit
```

```
/*------------------------- Subroutines --------------------------*/

SendFile:
/*----------------------------------------------------------------*/
/* Read the contents of the requested file and send each line of  */
/* the file to the partner program.                               */
/*----------------------------------------------------------------*/
address command 'EXECIO * DISKR' requested_file '(FINIS STEM LINE.'
do index = 1 to line.0
  buffer = line.index
  send_length = length(buffer)
  'CMSEND conversation_ID buffer send_length',
         'request_to_send_received return_code'
  say; say 'Routine called:  CMSEND'
  if (return_code ¬= CM_OK) then call ErrorHandler 'CMSEND'
  call TraceParms 'conversation_ID buffer send_length',
                  'request_to_send_received return_code'
end

return


TraceParms:
/*----------------------------------------------------------------*/
/* Display parameters and their values as passed to this subroutine.*/
/*----------------------------------------------------------------*/
parse arg parmlist
do word_num = 1 to words(parmlist)
  parameter = word(parmlist,word_num)
  select
    when (parameter = 'return_code') then
      say '  return_code is' cm_return_code.return_code
    when (parameter = 'buffer') then
      say '  buffer is' left(buffer,send_length)
    when (parameter = 'receive_buffer') then
      say '  buffer is' left(receive_buffer,received_length)
    when (parameter = 'data_received') then
      say '  data_received is' cm_data_received.data_received
    when (parameter = 'status_received') then
      say '  status_received is' cm_status_received.status_received
    when (parameter = 'request_to_send_received') then
      say '  request_to_send_received is',
          cm_request_to_send_received.request_to_send_received
    otherwise
      say ' ' parameter 'is' value(parameter)
  end
end

return


Error:
/*----------------------------------------------------------------*/
/* Report error when REXX special variable RC is not 0.           */
/*----------------------------------------------------------------*/
say
say '* ERROR:  REXX has detected an error'
say '          The return code variable RC was set to' rc
signal GetOut


ErrorHandler:
/*----------------------------------------------------------------*/
/* Report routine that failed and the error return code.          */
/*----------------------------------------------------------------*/
parse arg routine_name
```

```
say
say '* ERROR:  An error occurred during a' routine_name 'call'
say '          The return_code was set to' cm_return_code.return_code
signal GetOut
```

We will not rerun the programs for this change. If you do, the results for the last Receive call displayed at the SERVR terminal should report that neither data nor status was received, and the *return_code* parameter will be set to CM_DEALLOCATED_NORMAL. Because of our update to the SENDBACK EXEC, this *return_code* will no longer cause an error message to be displayed.

FYI: Receiving Partial Records

If you recall, when we added the Receive loop to the PROCESS EXEC in step 6, we only wrote the data being received to our output file when the *data_received* parameter had a value of CM_COMPLETE_DATA_RECEIVED. Because TEST FILE was composed of 80-character records and we specified 80 as the *requested_length* for the Receive call, the results always showed complete data being received. So, our program never had to receive just part of a line.

This works because both of our user IDs are on the same system. When the partners are on different systems, this is not likely to happen because of buffering at the LUs.

To see what would happen if partial records were received, you might want to temporarily change the *requested_length* to a lower number. A *requested_length* of 20, for example, will require that the Receive routine be called four times to completely receive 80 characters worth of data from the partner (on the same system). The first three calls to Receive will complete with a *data_received* value of CM_INCOMPLETE_DATA_RECEIVED, and on the fourth call, that parameter will be set to CM_COMPLETE_DATA_RECEIVED. If you check the OUTPUT LOGFILE, you will find that the partial records were correctly processed by the Receive loop.

In general, you should always write applications in such a way that they can handle partial records.

This brings us to the end of our introduction to the starter set of SAA CPI Communications routines. Now that you have built the sample programs and basically understand how they work together, you may find it beneficial to review the final summary section. It contains a flow diagram that shows how our sample programs would work in an SNA network.

In the next chapter we will begin covering some more advanced routines. We will be using the same two user IDs and adding further routine calls to our two execs, so please do not erase them. It might also be worthwhile to save a backup copy of each program at the end of each chapter.

Summary with Flow Diagram

Now that we see how the various CPI Communications starter set calls can be used to establish a conversation and exchange data, we can review what we have learned while seeing how the *Common Programming Interface Communications Reference* describes a conversation flow.

A Word about the Flow Diagrams

In the flow diagram we will be examining (Figure 24 on page 61), vertical dotted lines indicate the components involved in the exchange of information between systems. The horizontal arrows indicate the direction of the flow for that step. The numbers lined up on the left side of the flow are reference points to the flow and indicate the progression of the calls made on the conversation. These same numbers correspond to the numbers under the **Step** heading of the text description that follows.

The call parameter lists shown in the flows are not complete; only the parameters of particular interest to the flows being discussed are shown.

This flow diagram does not assume that both partners are on the same VM system. A complete discussion of all possible timing scenarios is beyond the scope of this book.

Flow Diagram for Starter Set Conversation

Figure 24 on page 61 shows the flow for the conversation developed in. this chapter.

The steps shown in Figure 24 on page 61 are:

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

Step	Description
5 and 6	PROCESS sends data with the Send_Data call and receives a *return_code* of CM_OK. Until now, the conversation may not have been established because the conversation startup request may not be sent until the first flow of data. In fact, any number of Send_Data calls can be issued before CPI Communications actually has a full buffer, which causes it to send the startup request and data. Step 5 shows a case where the amount of data sent by the first Send_Data is greater than the size of the local LU's send buffer (a system-dependent property), which is one of the conditions that triggers the sending of data. The request for a conversation is sent at this time.
7 and 8	After the conversation is established, the remote program's system takes care of starting SENDBACK. The conversation on SENDBACK's side is in **Reset** state and SENDBACK issues a call to Accept_Conversation, which places the conversation in **Receive** state. The Accept_Conversation call is similar to the Initialize_Conversation call in that it equates a *conversation_ID* with a set of conversation characteristics. SENDBACK, like PROCESS in Step 2, receives a unique *conversation_ID* that it will use in all future CPI Communications calls for that particular conversation.
9 and 10	After its end of the conversation is in **Receive** state, SENDBACK begins whatever processing role it and PROCESS have agreed upon. In this case, SENDBACK accepts data with a Receive call. PROCESS could continue to make Send_Data calls (and SENDBACK could continue to make Receive calls), but, for the purposes of our example, assume that PROCESS only wanted to send the data contained in its initial Send_Data call.
11	After sending some amount of data (an indeterminate number of Send_Data calls), PROCESS issues the Receive call while its end of the conversation is in **Send** state. This call causes the remaining data buffered at REQUESTR to be sent and permission to send to be given to SENDBACK. PROCESS's end of the conversation is placed in **Receive** state, and PROCESS waits for a response from SENDBACK. SENDBACK issues a Receive call in the same way it issued the previous Receive call.
12	SENDBACK receives not only the last of the data from PROCESS, but also a *status_received* parameter set to CM_SEND_RECEIVED. The value of CM_SEND_RECEIVED notifies SENDBACK that its end of the conversation is now in **Send** state.
13	As a result of the *status_received* value, SENDBACK issues a Send_Data call. The data from this call, on arrival at REQUESTR, is returned to PROCESS as a response to the Receive it issued in Step 11. At this point, the flow of data has been completely reversed and the two programs can continue whatever processing their logic dictates. To give control of the conversation back to PROCESS, SENDBACK would simply follow the same procedure that PROCESS executed in Step 11.
14 through 16	PROCESS and SENDBACK continue processing. SENDBACK sends data and PROCESS receives the data.
17	SENDBACK issues a Receive call from **Send** state to change its state back to **Receive**.
18	PROCESS receives the last of the data along with notification that SENDBACK has changed states.
19	PROCESS issues a Deallocate call to send any data buffered by the local system and release the conversation. The Receive call issued by SENDBACK in step 17 can now complete.
20 and 21	The *return_code* of CM_DEALLOCATED_NORMAL tells SENDBACK that the conversation is deallocated. Both SENDBACK and PROCESS finish normally. **Note:** Only one program should issue Deallocate; in this case it was PROCESS. If SENDBACK had issued Deallocate after receiving CM_DEALLOCATED_NORMAL, an error would have resulted.

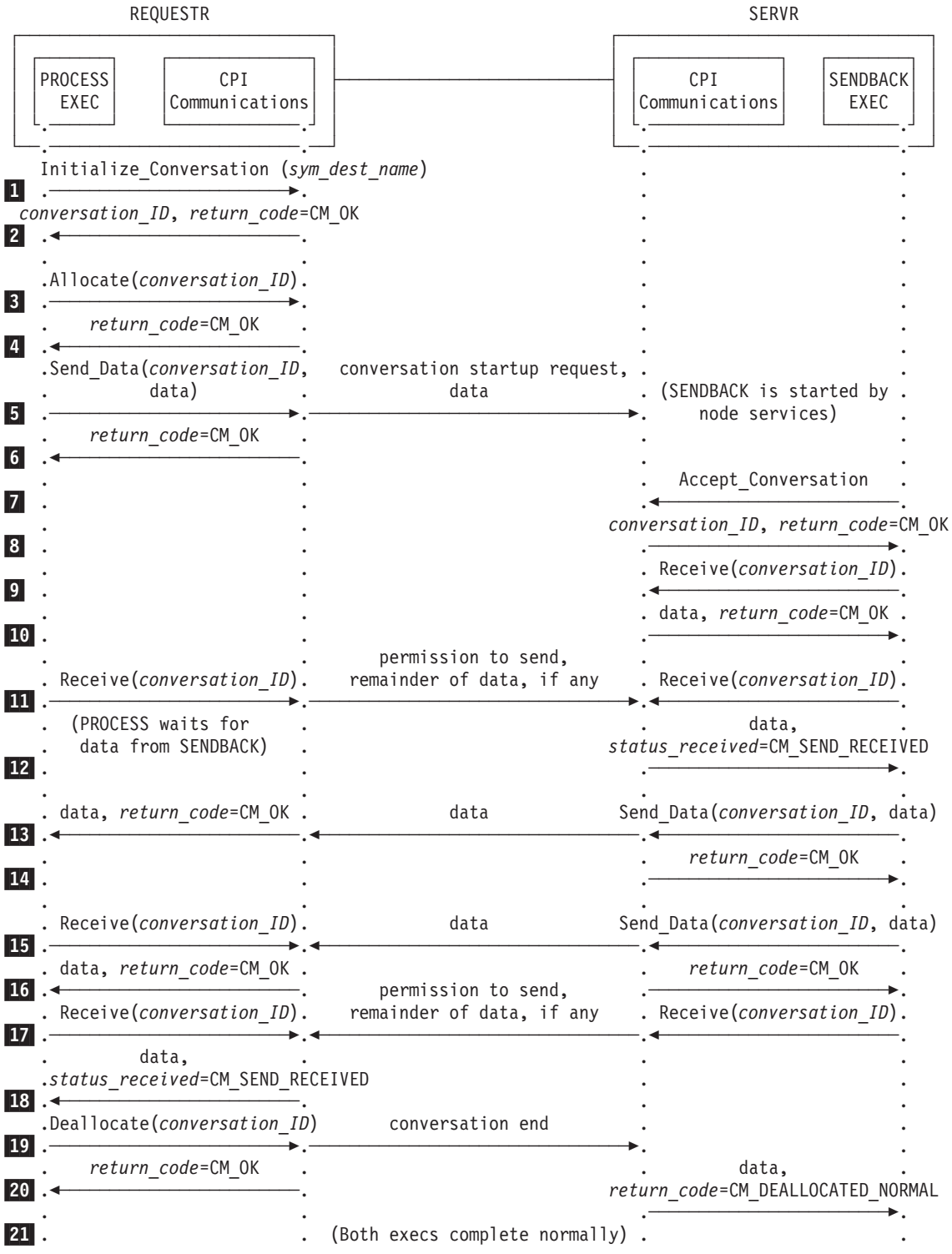


Figure 24. Flow Diagram for Starter Set Conversation

Chapter 3. Advanced CPI Communications Calls

In Chapter 2, “Starter Set CPI Communications Calls,” we developed a simple pair of communications programs to pass a file from one virtual machine to another. Communications programming is not always so straightforward, however. For situations demanding more flexibility, CPI Communications provides advanced-function calls that let programs modify conversation characteristics and synchronize activities between partners.

In this chapter we will discuss and use several of the advanced CPI Communications routines that are used for synchronization and control, for modifying, and for examining conversation characteristics.

Overview of Advanced CPI Communications Calls

You can use the advanced calls to do more specialized processing than is possible using the default set of characteristic values. The advanced calls provide more capabilities for synchronization and monitoring of data. For example, the Set calls let a program modify conversation characteristics, and the Extract calls let a program examine the conversation characteristics that have been assigned to a given conversation.

Note: Advanced CPI Communications calls can be used with the starter set calls, but are being introduced separately in this tutorial for the sake of simplicity.

The advanced function calls can be logically divided into three categories:
- Synchronization and control
- Modifying conversation characteristics
- Examining conversation characteristics.

The following tables list the calls in each category, giving both the pseudonyms and callable names.

Calls Used for Synchronization and Control

Pseudonym	Call	Description	Page
Confirm	CMCFM	Sends a confirmation and waits for a reply	80
Confirmed	CMCFMD	Sends a confirmation reply	81
Flush	CMFLUS	Explicitly sends any information held in the local send buffer	70
Prepare_To_Receive	CMPTR	Changes conversation state from **Send** to **Receive**	70
Request_To_Send	CMRTS	Sends notification to partner that local program has data to send	119
Send_Error	CMSERR	Notifies partner of an error that occurred during the conversation	107
Test_Request_To_Send_Received	CMTRTS	Determines whether partner program has requested to send data	122

Calls Used for Modifying Conversation Characteristics

Pseudonym	Call	Page
Set_Conversation_Type	CMSCT	109

Pseudonym	Call	Page
Set_Deallocate_Type	CMSDT	98
Set_Error_Direction	CMSED	120
Set_Fill	CMSF	120
Set_Log_Data	CMSLD	120
Set_Mode_Name	CMSMN	120
Set_Partner_LU_Name	CMSPLN	113
Set_Prepare_To_Receive_Type	CMSPTR	88
Set_Receive_Type	CMSRT	121
Set_Return_Control	CMSRC	121
Set_Send_Type	CMSST	93
Set_Sync_Level	CMSSL	74
Set_TP_Name	CMSTPN	116

Calls Used for Examining Conversation Characteristics

Pseudonym	Call	Page
Extract_Conversation_State	CMECS	65
Extract_Conversation_Type	CMECT	105
Extract_Mode_Name	CMEMN	119
Extract_Partner_LU_Name	CMEPLN	119
Extract_Sync_Level	CMESL	119

Using Advanced Set Calls

Table 2 shows in pseudocode style how we will be building on our programs in this chapter. The new calls we will be adding are denoted in boldface.

Table 2. Overview of Sample Programs with Advanced Set Calls

REQUESTR User ID	SERVR User ID

Initialize_Conversation
(Set_Conversation_Type)
Set_Partner_LU_Name
Set_TP_Name
Allocate
Send_Data
if performing confirmation
  Confirm
Set_Prepare_to_Receive_Type
Prepare_To-Receive
 	
-Receive loop-
do until send control returned
  Receive
  if confirmation requested
  Confirmed
 	
end
Deallocate
	Accept_Conversation
	Extract_Conversation_Type
	if conversation type is basic
	  Send_Error
	
	-Receive loop-
	do until notified of deallocation
	  Receive
	  if confirmation requested
	  Confirmed
	  if send control received
	
	  -Send loop-
	  do until all of file is sent
	  if last data record
	  Set_Send_Type
	  Send_Data
	  end
	
	end

The Set_Conversation_Type call is shown in parentheses in the table because we remove it after examining the consequences of its use. In addition, two more calls (**Set_Deallocate_Type** and **Extract_Conversation_State**) are included in subroutines that we will add to keep the flow a little cleaner.

You will notice that we have added calls to the programs we created in Chapter 2, "Starter Set CPI Communications Calls," on page 9. Each will be discussed and added to the execs we started in that chapter.

FYI: Tidying Up

It is time to clean up the PROCESS and SENDBACK execs by removing most of the parameters on the calls to our TraceParms subroutine. (You might want to make a backup copy of both execs before continuing.) Very carefully remove all the parameters from these calls **except** for the following two Receive parameters:

- `'data_received'`
- `'status_received'`.

In most cases, the call to TraceParms will be left with no parameters, but this will not hurt anything. As we add new calls to our program, we will be tracing other parameters in this chapter and we do not want the console log to be too long. From now on, we will remove extra parameters after we have seen the results of the call.

The ErrorHandler routine will continue to display any *return_code* values other than CM_OK or CM_DEALLOCATED_NORMAL. We will leave the `say` statements identifying the called routine.

Now, just to make sure you did not make any mistakes, after you have deleted the unwanted parameters, re-execute the execs to make sure they complete successfully. As you recall, you only need to execute the PROCESS EXEC, providing the parameter GETFILE as the symbolic destination name.

The Extract_Conversation_State (CMECS) Call

The Extract_Conversation_State (CMECS) call returns a value indicating the local program's current conversation state for a given conversation.

This routine is meant for use when a program is working with protected conversations (conversations with the *sync_level* characteristic set to CM_SYNC_POINT). It is also useful for debugging and error handling.

We can put the Extract_Conversation_State call to good use to help understand the concept of conversation states. But first, let's look at the parameters.

The format for Extract_Conversation_State is:

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

Input Parameter

Use the ***conversation_ID*** parameter to identify the conversation.

Output Parameters

The ***conversation_state*** parameter returns the current state of the conversation identified by the input *conversation_ID* parameter. Possible values for this characteristic are:

- CM_INITIALIZE_STATE (2)
- CM_SEND_STATE (3)
- CM_RECEIVE_STATE (4)
- CM_SEND_PENDING_STATE (5)
- CM_CONFIRM_STATE (6)
- CM_CONFIRM_SEND_STATE (7)
- CM_CONFIRM_DEALLOCATE_STATE (8)
- CM_DEFER_RECEIVE_STATEv (9)
- CM_DEFER_DEALLOCATE_STATE (10)
- CM_SYNC_POINT_STATE (11)
- CM_SYNC_POINT_SEND_STATE (12)
- CM_SYNC_POINT_DEALLOCATE_STATE (13)

See Appendix B, “CPI Communications Conversation States,” on page 223 for more information on the various conversation states.

The ***return_code*** parameter is a variable for returning the result of the call execution. Possible values of interest to us are:

CM_OK (0)
: indicates that the conversation state has been extracted.

CM_PRODUCT_SPECIFIC_ERROR (20)
: indicates a CMS error; check the CPICOMM LOGDATA file for a summary of the error.

CM_PROGRAM_PARAMETER_CHECK (24)
: indicates that the specified conversation ID is unassigned.

Results of the Call

Anything other than a return code of CM_OK yields a *conversation_state* value that is undefined and should not be examined. This call does not cause a state change.

Adding CMECS to Both Our Programs

By adding an Extract_Conversation_State call to the TraceParms subroutine in both of our programs, we will be able to monitor the conversation states on each side of the conversation.

Let’s add the new call. The only change is in the TraceParms subroutine, which currently is identical in both of our execs, so we are just showing that section of the program.

The TraceParms subroutine in **both** execs (PROCESS and SENDBACK) now contains these lines:

```
   ⋮
TraceParms:
/*-----------------------------------------------------------------*/
/* Display parameters and their values as passed to this subroutine.*/
/*-----------------------------------------------------------------*/
parse arg parmlist
do word_num = 1 to words(parmlist)
  parameter = word(parmlist,word_num)
  select
    when (parameter = 'return_code') then
      say '  return_code is' cm_return_code.return_code
    when (parameter = 'buffer') then
      say '  buffer is' left(buffer,send_length)
```

```
    when (parameter = 'receive_buffer') then
      say '  buffer is' left(receive_buffer,received_length)
    when (parameter = 'data_received') then
      say '  data_received is' cm_data_received.data_received
    when (parameter = 'status_received') then
      say '  status_received is' cm_status_received.status_received
    when (parameter = 'request_to_send_received') then
      say '  request_to_send_received is',
          cm_request_to_send_received.request_to_send_received
    otherwise
      say ' ' parameter 'is' value(parameter)
  end
end
/*--------------------------------------------------------------------*/
/* Extract the current conversation state of the local program.     */
/*--------------------------------------------------------------------*/
'CMECS conversation_ID conversation_state return_code'
if (return_code = CM_OK) then
  say '  conversation_state is =>',
      cm_conversation_state.conversation_state

return
   :
```

The Flow of a Conversation

This time as we review the progression of the conversation between our programs, we will break the displayed output into sections, like snapshots of the conversation. We want to watch for the relationship between the routine calls, the call results, and the conversation states. Enter

```
process getfile
```

from the REQUESTR user ID and we will proceed.

Looking at the requester's side of the conversation first, we see the following lines displayed at the REQUESTR terminal:

```
process getfile
Requesting the file:  TEST FILE A

Routine called:  CMINIT
  conversation_state is => CM_INITIALIZE_STATE

Routine called:  CMALLC
  conversation_state is => CM_SEND_STATE
   :
```

Figure 25. Results of First Two Calls from PROCESS EXEC

The requester's first call is to Initialize_Conversation. The appropriate communications directory is checked for side information and default values are set. Upon completion of the Initialize_Conversation call, a conversation identifier is returned to the program and the state of the conversation is changed from **Reset** to **Initialize** state.

Initialize state can be considered a transition state. The program can now issue Extract calls to view conversation characteristics and Set calls to override default characteristics or values obtained from side information. So far, though, this is a very one-sided conversation.

After the Allocate call is issued, a connection (session in SNA communications terminology) is established between the local and remote systems, if one does not already exist, over which the conversation will flow. Then the conversation state changes to **Send** state. The program can now send data on the conversation.

```
   .
   .
   .
Routine called:  CMSEND
  conversation_state is => CM_SEND_STATE

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED
  conversation_state is => CM_RECEIVE_STATE
   .
   .
   .
```

Figure 26. Results of Next Two Calls from PROCESS EXEC

The requester sends the name of the file it is requesting to the server, and the conversation remains in **Send** state. The side of the conversation in **Send** state maintains control of the conversation until it changes the state or an error occurs.

This requester, in fact, does want to change states, which it accomplishes by calling Receive. That action allows the partner to send data, which the requester then receives. Not surprisingly, the requester's side of the conversation has entered **Receive** state.

```
   .
   .
   .
Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED
  conversation_state is => CM_RECEIVE_STATE

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_SEND_RECEIVED
  conversation_state is => CM_SEND_STATE
   .
   .
   .
```

Figure 27. Results of Next Two Receive Calls from PROCESS EXEC

Data reception continues. The last Receive call shown completes with *status_received* of CM_SEND_RECEIVED. That status value switches the conversation state back to **Send** state. The requester again controls the conversation.

```
   .
   .
   .
Routine called:  CMDEAL
Ready;
```

Figure 28. Completion of PROCESS EXEC Execution

Having received the file from the server, the requester can terminate the conversation by issuing a Deallocate call. As a result of deallocation, the conversation identifier becomes unassigned. If the requester program tries to extract the conversation state after the Deallocate, *return_code* would be set to CM_PROGRAM_PARAMETER_CHECK because the specified conversation

identifier no longer has any meaning. In addition, because the *return_code* is not CM_OK, the *conversation_state* parameter value is undefined.

The conversation has returned to **Reset** state.

Viewing the conversation from the SERVR user ID, we will see:

```
Routine called:  CMACCP
  conversation_state is => CM_RECEIVE_STATE
   ⋮
```

Figure 29. Results of First Call from SENDBACK EXEC

The private server program is started and an Accept_Conversation call is made, taking the server's end of the conversation from **Reset** to **Receive** state. A conversation identifier is also returned for the server. This conversation ID is not related to the conversation ID on the other end of the conversation.

```
   ⋮
Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED
  conversation_state is => CM_RECEIVE_STATE

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_SEND_RECEIVED
  conversation_state is => CM_SEND_STATE
   ⋮
```

Figure 30. Results of Next Two Calls from SENDBACK EXEC

The first Receive issued completes with receipt of the file name sent by the partner. The conversation remains in **Receive** state until *status_received* is returned with a value of CM_SEND_RECEIVED. The partner has given the server program control of the conversation, which results in the conversation state change to **Send** state.

```
   ⋮
Routine called:  CMSEND
  conversation_state is => CM_SEND_STATE

Routine called:  CMSEND
  conversation_state is => CM_SEND_STATE

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Conversation deallocated by partner
Ready;
```

Figure 31. Completion of SENDBACK EXEC Execution

After responding to the requester by sending the contents of the requested file, the server application calls Receive to return control to the requester. Although we did not display it, the *return_code* on the last Receive call was CM_DEALLOCATED_NORMAL, indicating that the partner deallocated the conversation. (Notice that our correction to the server program in the previous

chapter avoided an error message for this *return_code* value.) The conversation identifier for the server's side of the conversation becomes unassigned, and the conversation enters **Reset** state.

FYI: Flush (CMFLUS) Call Overview

The Flush (CMFLUS) call empties the send buffer of the local system (logical unit or LU, meaning the node in the SNA network) for a given conversation. When notified by CPI Communications that a Flush has been issued, the local LU sends any information it has buffered to the remote LU. The buffered information can come from Allocate (CMALLC), Send_Data (CMSEND), and Send_Error (CMSERR) calls.

To optimize transmissions between the conversation partners, the local LU typically buffers the data from consecutive Send_Data calls until the local buffer is full. The amount of data sufficient for transmission depends on the characteristics of the session allocated for the conversation and may vary from one session to another.

Using the Flush call can improve application performance when data in the local buffer is needed by the partner for immediate processing. Also, issuing Flush immediately after an Allocate call should ensure that the partner program is started as soon as possible.

If the local LU has no information in its send buffer, nothing is transmitted to the remote LU when Flush is called.

In general, however, Flush should be used sparingly. There is no need to call it if the data is not required immediately by the partner program. If you need to be sure your partner gets data or allocation information immediately, however, and you are not changing states, it may be appropriate to call Flush so your partner can begin processing.

We will not be adding the Flush call to either of our programs because it would have no effect in our example scenario, but we wanted to introduce this routine because several other CPI Communications routines that we will be discussing can perform implicit flushes as part of their processing.

As we saw when we added the Extract_Conversation_State (CMECS) call to our program, issuing a successful Receive (CMRCV) call from **Send** state really does switch the local end of the conversation to **Receive** state. Another way to accomplish that state change is to call the Prepare_To_Receive (CMPTR) routine.

The Prepare_To_Receive (CMPTR) Call

The Prepare_To_Receive (CMPTR) call changes a conversation from **Send** to **Receive** state in preparation for receiving data. As a result of the Prepare_To_Receive call, the local LU's send buffer may be flushed.

One advantage of the Prepare_To_Receive call is that the calling program is not held up waiting for the partner to respond with data or status, as would be the case with the Receive call.

The format for Prepare_To_Receive is:

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

Input Parameter

Use the ***conversation_ID*** parameter to identify the conversation.

Output Parameter

The possible values for the ***return_code*** parameter that are of interest to us are:

CM_OK (0)
: indicates that the Prepare_To_Receive call completed successfully.

CM_PRODUCT_SPECIFIC_ERROR (20)
: indicates a CMS error; check the CPICOMM LOGDATA file for a summary of the error.

CM_PROGRAM_PARAMETER_CHECK (24)
: indicates that the specified conversation ID is unassigned.

CM_PROGRAM_STATE_CHECK (25)
: can indicate several problems, but the most common error would be that the program is not in **Send** or **Send-Pending** state.

Results of the Call

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

Adding CMPTR to Our Requester Program

When programming with CPI Communications, several ways may be available to implement the same function. For example, we can add a Prepare_To_Receive call to our requester program to change the requester's side of the conversation to **Receive** state rather than letting one of the Receive (CMRCV) calls do that.

Let's add the Prepare_To_Receive call to the PROCESS EXEC, immediately following the Send_Data (CMSEND) call.

Note: As you add new code, remember to remove the extra parameters (keeping only *data_received* and *status_received*) from the TraceParms calls from the previous addition. The error routine will continue to display any *return_code* values other than CM_OK or CM_DEALLOCATED_NORMAL.

Your exec should now have the following lines in it:

```
/*====================================================================*/
/*  PROCESS EXEC - Sample file requester application.                 */
/*====================================================================*/

  ⋮
/*--------------------------------------------------------------------*/
/* Send the name of the file being requested to the partner program.*/
/*--------------------------------------------------------------------*/
buffer = fname ftype fmode
send_length = length(buffer)
'CMSEND conversation_ID buffer send_length',
       'request_to_send_received return_code'
say; say 'Routine called:  CMSEND'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMSEND'
call TraceParms
/*--------------------------------------------------------------------*/
/* Issue Prepare_To_Receive to switch the conversation state from   */
/* Send state to Receive state.                                       */
/*--------------------------------------------------------------------*/
'CMPTR conversation_ID return_code'
```

```
say; say 'Routine called:  CMPTR'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMPTR'
call TraceParms 'conversation_ID return_code'
/*-------------------------------------------------------------------*/
/* Start a Receive loop.  Receive calls will be issued until         */
/* notification that the partner has finished sending data and       */
/* entered Receive state at its end of the conversation (noted by    */
/* receipt of CM_SEND_RECEIVED                                       */
/* for status_received) or until a return_code value other than      */
/* CM_OK is returned.  The record length of the incoming data        */
/* is assumed to be 80 bytes, or less.                               */
/*-------------------------------------------------------------------*/
complete_line = ''
requested_length = 80
do until (status_received = CM_SEND_RECEIVED)
  /*-----------------------------------------------------------------*/
  /* Receive information from the conversation partner.              */
  /*-----------------------------------------------------------------*/
  'CMRCV conversation_ID receive_buffer requested_length',
        'data_received received_length status_received',
        'request_to_send_received return_code'
  .
  .
  .
```

After filing the exec and entering

```
process getfile
```

the REQUESTR virtual machine results will be:

```
process getfile
Requesting the file:  TEST FILE A

Routine called:  CMINIT
  conversation_state is => CM_INITIALIZE_STATE

Routine called:  CMALLC
  conversation_state is => CM_SEND_STATE

Routine called:  CMSEND
  conversation_state is => CM_SEND_STATE

Routine called:  CMPTR
  conversation_ID is 00000000
  return_code is CM_OK
  conversation_state is => CM_RECEIVE_STATE

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED
  conversation_state is => CM_RECEIVE_STATE

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED
  conversation_state is => CM_RECEIVE_STATE

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_SEND_RECEIVED
  conversation_state is => CM_SEND_STATE

Routine called:  CMDEAL
Ready;
```

Figure 32. Execution Results after Adding CMPTR to PROCESS EXEC

And, the SERVR virtual machine results will be:

```
Routine called:  CMACCP
  conversation_state is => CM_RECEIVE_STATE

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED
  conversation_state is => CM_RECEIVE_STATE

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_SEND_RECEIVED
  conversation_state is => CM_SEND_STATE

Routine called:  CMSEND
  conversation_state is => CM_SEND_STATE

Routine called:  CMSEND
  conversation_state is => CM_SEND_STATE

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Conversation deallocated by partner
Ready;
```

Figure 33. Results from SENDBACK EXEC Execution

The same number of Receive calls are being issued now as before the Prepare_To_Receive call was added, but the first Receive issued by the requester is no longer serving the dual purpose of changing the conversation state and receiving data.

Notice that the conversation state following the Prepare_To_Receive call is **Receive** state.

The Set_Sync_Level (CMSSL) Call

The Set_Sync_Level (CMSSL) call sets the *sync_level* characteristic for a given conversation and overrides the sync level assigned with the Initialize_Conversation (CMINIT) call.

The *sync_level* characteristic specifies the level of synchronization processing between the two programs. It determines whether the programs support no synchronization, confirmation-level synchronization, or sync-point-level synchronization.

Only a program initiating a conversation (using the Initialize_Conversation call) can issue the Set_Sync_Level call. The call must be issued while in **Initialize** state, prior to the Allocate (CMALLC) call for the specified conversation.

The format for Set_Sync_Level is:

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

Input Parameters

Use the ***conversation_ID*** parameter to identify the conversation for which the *sync_level* characteristic is to be changed.

Use the ***sync_level*** parameter to specify the synchronization level that the local and remote programs can use on the conversation. This characteristic can be set to one of the following values:

CM_NONE (0)
: No confirmation processing will occur on this conversation. The programs will neither issue nor recognize any synchronization requests.

CM_CONFIRM (1)
: Confirmation processing can be performed on this conversation. The programs can issue calls and recognize returned parameters relating to confirmation.

CM_SYNC_POINT (2)
: The programs can perform sync point processing on this conversation. The programs can issue calls to a synchronization point service, will recognize returned parameters relating to sync point processing, and can perform confirmation processing.

  The use of CM_SYNC_POINT to synchronize the committing and backing out of data updates is beyond the scope of this book.

Output Parameter

Possible values for the ***return_code*** parameter are:

CM_OK (0)
: indicates that the *sync_level* value has been changed.

CM_PRODUCT_SPECIFIC_ERROR (20)
: indicates a CMS error; check the CPICOMM LOGDATA file.

CM_PROGRAM_PARAMETER_CHECK (24)
: indicates that the specified conversation ID is unassigned, the *sync_level* is set to an undefined value, the specified *sync_level* conflicts with another conversation characteristic, or the *sync_level* is set to CM_SYNC_POINT and the local system does not support a synchronization point service.

CM_PROGRAM_STATE_CHECK (25)
: indicates that the conversation is not in **Initialize** state.

Results of the Call

A return code other than CM_OK results in no change to the *sync_level* characteristic. This call does not cause a state change.

Adding CMSSL to Our Requester Program

Setting the *sync_level* to CM_NONE would have no effect on our program because that is the default value assigned with the Initialize_Conversation call. We will not be discussing sync point processing, so we will set the *sync_level* to CM_CONFIRM.

Confirmation processing provides a chance for the program receiving data to let the sender know whether the data is getting through and being processed. You can use confirmation processing to provide checkpoints in an application.

Let's add the Set_Sync_Level call to the requester application, immediately following the Initialize_Conversation call. We will also add a console prompt so that we can decide at run time whether we want to enable confirmation processing.

The PROCESS EXEC should now have the following lines:

```
/*====================================================================*/
/*  PROCESS EXEC - Sample file requester application.                 */
/*====================================================================*/
```

```
   ⋮
/*--------------------------------------------------------------------*/
/* Initialize the conversation.                                       */
/*--------------------------------------------------------------------*/
'CMINIT conversation_ID sym_dest_name return_code'
say; say 'Routine called:  CMINIT'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMINIT'
call TraceParms
/*--------------------------------------------------------------------*/
/* Determine if confirmation processing is desired.                   */
/*--------------------------------------------------------------------*/
say; say 'Would you like confirmation processing? (Y/N)'
parse upper pull perform_confirm
if (perform_confirm = 'Y') then
  do
    /*----------------------------------------------------------------*/
    /* Set sync_level to CM_CONFIRM.                                  */
    /*----------------------------------------------------------------*/
    sync_level = CM_CONFIRM
    'CMSSL conversation_ID sync_level return_code'
    say; say 'Routine called:  CMSSL'
    if (return_code ¬= CM_OK) then call ErrorHandler 'CMSSL'
    call TraceParms 'conversation_ID sync_level return_code'
    say '  Confirmation processing enabled'
  end
/*--------------------------------------------------------------------*/
/* Allocate the conversation.                                         */
/*--------------------------------------------------------------------*/
   ⋮
/*------------------------- Subroutines ------------------------------*/

TraceParms:
/*--------------------------------------------------------------------*/
/* Display parameters and their values as passed to this subroutine.*/
/*--------------------------------------------------------------------*/
parse arg parmlist
do word_num = 1 to words(parmlist)
  parameter = word(parmlist,word_num)
  select
    when (parameter = 'return_code') then
      say '  return_code is' cm_return_code.return_code
    when (parameter = 'buffer') then
      say '  buffer is' left(buffer,send_length)
    when (parameter = 'receive_buffer') then
      say '  buffer is' left(receive_buffer,received_length)
    when (parameter = 'data_received') then
      say '  data_received is' cm_data_received.data_received
    when (parameter = 'status_received') then
      say '  status_received is' cm_status_received.status_received
    when (parameter = 'request_to_send_received') then
      say '  request_to_send_received is',
          cm_request_to_send_received.request_to_send_received
    when (parameter = 'sync_level') then
      say '  sync_level is' cm_sync_level.sync_level
    otherwise
      say ' ' parameter 'is' value(parameter)
  end
end
   ⋮
```

After filing the exec, enter

```
process getfile
```

and choose confirmation processing when the prompt is displayed.

The results on the REQUESTR user ID should be:

```
process getfile
Requesting the file:  TEST FILE A

Routine called:  CMINIT
  conversation_state is => CM_INITIALIZE_STATE

Would you like confirmation processing? (Y/N)
Y

Routine called:  CMSSL
  conversation_ID is 00000000
  sync_level is CM_CONFIRM
  return_code is CM_OK
  conversation_state is => CM_INITIALIZE_STATE
  Confirmation processing enabled

Routine called:  CMALLC
  conversation_state is => CM_SEND_STATE

Routine called:  CMSEND
  conversation_state is => CM_SEND_STATE

Routine called:  CMPTR

* ERROR:  An error occurred during a CMPTR call
          The return_code was set to CM_RESOURCE_FAILURE_NO_RETRY
Ready;
```

Figure 34. Results of Adding CMSSL Call to PROCESS EXEC

The results on the SERVR user ID will be:

```
Routine called:  CMACCP
  conversation_state is => CM_RECEIVE_STATE

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED
  conversation_state is => CM_RECEIVE_STATE

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_CONFIRM_SEND_RECEIVED
  conversation_state is => CM_CONFIRM_SEND_STATE

Routine called:  CMRCV

* ERROR:  An error occurred during a CMRCV call
          The return_code was set to CM_PROGRAM_STATE_CHECK
Ready;
```

Figure 35. Results from SENDBACK EXEC Execution

Even though our Set_Sync_Level call executed correctly, a subsequent call to Prepare_To_Receive (CMPTR) failed with *return_code* set to CM_RESOURCE_FAILURE_NO_RETRY.

The requester enabled confirmation processing, but the server program was not prepared to handle that situation. We can look at confirmation processing as a pause in the passing of data during which an exchange of confirmation information

takes place. It is like an “aside” or a very short conversation within the main conversation to make sure both partners are at the point in processing where they are expected to be.

As it turns out, choosing confirmation processing by setting the *sync_level* characteristic to CM_CONFIRM has implications for the Deallocate (CMDEAL) and Prepare_To_Receive (CMPTR) calls. In fact, if not otherwise changed by the specific Set calls, the default *deallocate_type* and *prepare_to_receive_type* values (CM_DEALLOCATE_SYNC_LEVEL and CM_PREP_TO_RECEIVE_SYNC_LEVEL, respectively) dictate the type of processing the Deallocate and Prepare_To_Receive calls perform based on the sync level of the conversation.

What Prepare_To_Receive does when the *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and the *sync_level* characteristic is CM_CONFIRM is to ask the partner to confirm that whatever data has been sent to the partner has been received and processed by the partner. This results in a change to the state of the conversation. After receiving the *status_received* value of CM_CONFIRM_SEND_RECEIVED, the server is in **Confirm-Send** state, but SENDBACK EXEC was not expecting to be in this state.

The Receive loop in SENDBACK EXEC did not check for a *status_received* value of CM_CONFIRM_SEND_RECEIVED, and so continued executing by calling Receive (CMRCV) again. Receive, however, cannot be called from **Confirm-Send** state, so the call completed with a *return_code* of CM_PROGRAM_STATE_CHECK and the program was terminated. The ending of the server application was in turn reflected to the requester by the *return_code* value of CM_RESOURCE_FAILURE_NO_RETRY on the Prepare_To_Receive (CMPTR) call.

With confirmation processing we find that the conversation states we have been using thus far are no longer adequate. Although all that probably seems a bit confusing, it is not as bad as it may seem at first. The next section will help you to determine which calls you can make from a particular state.

The State Table–Finding Out Where You Can Go from Here

As you will recall from our discussion of a two-way radio conversation, states are important to CPI Communications because they help us to synchronize our activities with those of our partner. Although the concept of conversation states is helpful, the enforcement of that concept is what helps us write programs that work correctly. As we have seen, it can get complicated trying to keep track of all the states and which calls we can make from a given state at a particular time.

The state that a conversation is going to enter next can be readily determined by examining the state table included as an appendix in the *Common Programming Interface Communications Reference*. The state table also indicates which CPI Communications calls can be issued while in a particular conversation state.

For a brief introduction to the state table, we have combined sections from several real entries into a modified (and simplified) table of our own. Our table steps through the part of SENDBACK EXEC’s side of the conversation that we just discussed, as it could be viewed from the state table’s perspective. As shown in Table 3 on page 79, the routines called by SENDBACK EXEC appear on the left side of our table, and the conversation states appear along the top. The number shown with the state name at the top of the column corresponds to the integer value of that state.

Table 3. State Transitions for SENDBACK EXEC CPI Communications Calls

Inputs	Reset 1	Initialize 2	Send 3	Receive 4	Confirm-Send 7
Accept_Conversation	↓	/	/	/	/
[ok]	4				
Receive(W)	[pc]	[sc]	↓′	↓′	[sc]
[ok] {dr,no}			4	–	
Receive(W)	[pc]	[sc]	↓′	↓′	[sc]
[ok] {*,cs}			7	7	
Confirmed	[pc]	[sc]	[sc]	[sc]	↓′
[ok]					3

The symbols and abbreviations that are used are all explained in the section preceding the state table in the *Common Programming Interface Communications Reference*. We will discuss only those that we need for this example.

When the SENDBACK EXEC is started, no conversation exists, so the conversation is said to be in **Reset** state, and the first routine that gets called is Accept_Conversation. To determine what happens when Accept_Conversation is called from **Reset** state, we need to examine the intersection of the corresponding row and column of the state table. In our example, we find the symbol "↓", which indicates that it is valid to call the specified routine while the conversation is in the state shown at the top of the column. The "/" in the next column indicates that it is impossible to call Accept_Conversation from **Initialize** state, which seems reasonable because the accepting side of a conversation never enters **Initialize** state.

Next, we want to check the state transition. The Accept_Conversation call completes with a *return_code* of CM_OK, which is represented in the state table with the symbol "[ok]". By looking in the column of the state we are currently in (**Reset**) and on the row reflecting the results of the call ([ok] under Accept_Conversation), we can determine what state transition will occur. In our case, we find a "4", the integer value corresponding to **Receive** state. As we have already seen, the conversation did in fact enter **Receive** state following the completion of the Accept_Conversation call.

The rest of the conversation can be traced through the state table in a similar fashion. We issue a Receive call from **Receive** state that completes with a *return_code* of CM_OK and the receipt of data but not status ({dr,no}). The table shows a "–", indicating that the conversation remains in the current state of **Receive** state.

A second Receive is issued, and it successfully completes with no data but a *status_received* of CM_CONFIRM_SEND_RECEIVED ([ok] {*,cs}). The asterisk, in this case, means CM_NO_DATA_RECEIVED. The "7" means that the conversation now enters **Confirm-Send** state. This is where our program runs into trouble. SENDBACK is expecting to receive either data or a *status_received* value of CM_SEND_RECEIVED. Because the *status_received* value does not match, the loop is executed again resulting in another Receive call. Having entered **Confirm-Send** state, however, this end of the conversation is now expected to either confirm that it has received and processed the data sent by its partner by issuing a Confirmed (CMCFMD) call, or let its partner know that there is a problem by issuing Send_Error (CMSERR) or Deallocate (CMDEAL) with the *deallocate_type* set to CM_DEALLOCATE_ABEND. As the Receive call rows of the

table show, calling Receive from **Confirm-Send** state results in a state check [sc], which was reflected to SENDBACK EXEC by the *return_code* value of CM_PROGRAM_STATE_CHECK.

For our purposes, it is adequate simply to issue a Confirmed call to tell the requester side that we have received and processed the data successfully. We will be discussing the Confirmed call shortly, but let's examine what effect that call will have while we are looking at our example state table. If we add a Confirmed call after the Receive, a *return_code* of CM_OK upon completion of that Confirmed call will indicate that the conversation has entered state "3", **Send** state. At that point, SENDBACK will have send control for the conversation and will be free to send the contents of the requested file.

Before we add a Confirmed call to our server application, though, let's continue our discussion of confirmation processing and what that means to both sides of the conversation.

Confirmation Processing

Now, let's continue our introduction to confirmation-level synchronization with the Confirm (CMCFM) and Confirmed (CMCFMD) calls. Because Confirmed (CMCFMD) is used as a response to Confirm (CMCFM), we will cover both routines before adding them to our programs.

The Confirm (CMCFM) Call

A program uses the Confirm (CMCFM) call to send a confirmation request to its partner program and then to wait for a reply. If all is well, the partner responds with a Confirmed (CMCFMD) call. These two calls working together can help programs synchronize their processing of data.

The program can call the Confirm routine only when the conversation associated with the specified *conversation_ID* has its *sync_level* characteristic set to CM_CONFIRM or CM_SYNC_POINT.

Like Flush (CMFLUS), Confirm is another call that should be used only when it is necessary, because it could adversely affect a program's performance. Because the program that issues Confirm must wait for a reply from its partner, the calling program's processing is suspended while it waits. If the partner fails to respond, the program that issued Confirm is left waiting indefinitely.

A common use of Confirm is to verify that the partner has received, validated, or processed data that was sent to it. Confirmed would be the affirmative response. If the remote program detects an error, it can give a negative response by issuing Send_Error (CMSERR) or Deallocate (CMDEAL) with *deallocate_type* set to CM_DEALLOCATE_ABEND.

The format for Confirm is:

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

Input Parameter

Use the ***conversation_ID*** parameter to identify the conversation.

Output Parameters

The ***request_to_send_received*** parameter returns an indication of whether a request-to-send notification has been received from the partner program. Possible values are:

CM_REQ_TO_SEND_NOT_RECEIVED (0)
CM_REQ_TO_SEND_RECEIVED (1)

If a request-to-send notification was received, it means that the remote program has requested that the local program's end of the conversation enter **Receive** state, which would place the remote program's end of the conversation in **Send** state.

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

The ***return_code*** values of interest to us are:

CM_OK (0)
: indicates that the remote program replied Confirmed (CMCFMD).

CM_DEALLOCATED_ABEND (17)
: usually indicates that the remote program deallocated the conversation with *deallocate_type* set to CM_DEALLOCATE_ABEND, or the remote LU did so because of a remote program abnormal-ending condition. The conversation is in **Reset** state.

CM_PRODUCT_SPECIFIC_ERROR (20)
: indicates a CMS error; check the CPICOMM LOGDATA file.

CM_PROGRAM_ERROR_PURGING (22)
: indicates that the remote program issued a Send_Error call and the conversation for the remote program was in **Receive** or **Confirm** state. The call may have caused information to be purged. Purging occurs when the remote program issues Send_Error for a conversation in **Receive** state before receiving all the information that the local program sent (all of the information sent before the CM_PROGRAM_ERROR_PURGING return code was reported to the local program).

CM_PROGRAM_PARAMETER_CHECK (24)
: indicates that the specified conversation ID is unassigned or that the *sync_level* conversation characteristic is set to CM_NONE.

CM_PROGRAM_STATE_CHECK (25)
: usually indicates that the conversation is not in **Send** or **Send-Pending** state.

Allocation errors can also be returned on a Confirm call.

Results of the Call

When the return code is CM_OK (0):

- No state change occurs if the program that issued the call was already in **Send** state.
- The conversation enters **Send** state if the program issued the call when the conversation was in **Send-Pending** state.

The Confirmed (CMCFMD) Call

A program uses the Confirmed (CMCFMD) call to send a confirmation reply to its partner program. The local program must have received a confirmation request before it can issue this call.

The format for Confirmed is:

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

Input Parameter

Use the ***conversation_ID*** parameter to identify the conversation.

Output Parameter

Possible values for the ***return_code*** parameter are:

CM_OK (0)
: indicates that a confirmation reply has been sent to the partner program.

CM_PRODUCT_SPECIFIC_ERROR (20)
: indicates a CMS error; check the CPICOMM LOGDATA file.

CM_PROGRAM_PARAMETER_CHECK (24)
: indicates that the specified conversation ID is unassigned.

CM_PROGRAM_STATE_CHECK (25)
: indicates that the conversation is not in **Confirm**, **Confirm-Send**, or **Confirm-Deallocate** state.

Results of the Call

A CM_OK return code affects the program state as follows:

- The conversation returns to **Receive** state if the program was in **Confirm** state (received CM_CONFIRM_RECEIVED in the *status_received* parameter on the preceding Receive call).
- The conversation enters **Send** state if the program was in **Confirm-Send** state (received CM_CONFIRM_SEND_RECEIVED in the *status_received* variable on the preceding Receive call).
- The conversation enters **Reset** state if the program was in **Confirm-Deallocate** state (received CM_CONFIRM_DEALLOC_RECEIVED in the *status_received* variable on the preceding Receive call).

Adding CMCFM and CMCFMD to Our Programs

Let's add the Confirm call to the requester application immediately following the Send_Data call that is issued to send the name of the file we are requesting. We will add the Confirmed call to the Receive loop in the server application.

When the Confirm call completes successfully, we will know that the partner application has started and that it has received the data. For our simple example, we will not be validating the received data. Both programs will simply be responding with Confirmed when any confirmation request is received.

Remember that the *sync_level* of CM_CONFIRM can affect the Prepare_To_Receive (CMPTR) and Deallocate (CMDEAL) calls by making them wait until the partner responds.

When confirmation processing is enabled, both programs will need to check for confirmation requests. We encountered a problem with our programs the last time we executed them for that reason, so we will also want to add a Confirmed call inside the requester's Receive call loop so the program can handle confirmation processing.

The requester's PROCESS EXEC should now have the following lines:

```
/*================================================================*/
/*  PROCESS EXEC - Sample file requester application.             */
/*================================================================*/
```

```
   ⋮
/*-------------------------------------------------------------------*/
/* Send the name of the file being requested to the partner program.*/
/*-------------------------------------------------------------------*/
buffer = fname ftype fmode
send_length = length(buffer)
'CMSEND conversation_ID buffer send_length',
       'request_to_send_received return_code'
say; say 'Routine called:  CMSEND'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMSEND'
call TraceParms
/*-------------------------------------------------------------------*/
/* Confirm that partner has started and received the name of        */
/* the requested file.                                              */
/*-------------------------------------------------------------------*/
'CMCFM conversation_ID request_to_send_received',
      'return_code'
say; say 'Routine called:  CMCFM'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMCFM'
call TraceParms 'conversation_ID request_to_send_received',
                'return_code'
/*-------------------------------------------------------------------*/
/* Issue Prepare_To_Receive to switch the conversation state from   */
/* Send state to Receive state.                                     */
/*-------------------------------------------------------------------*/
'CMPTR conversation_ID return_code'
say; say 'Routine called:  CMPTR'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMPTR'
call TraceParms
/*-------------------------------------------------------------------*/
/* Start a Receive loop.  Receive calls will be issued until        */
/* notification that the partner has finished sending data and      */
/* entered Receive state at its end of the conversation (noted by   */
/* receipt of CM_SEND_RECEIVED or CM_CONFIRM_SEND_RECEIVED          */
/* for status_received) or until a return_code value other than     */
/* CM_OK is returned.  The record length of the incoming data       */
/* is assumed to be 80 bytes, or less.                              */
/*-------------------------------------------------------------------*/
complete_line = ''
requested_length = 80
do until (status_received = CM_SEND_RECEIVED) |,
         (status_received = CM_CONFIRM_SEND_RECEIVED)
  /*-----------------------------------------------------------------*/
  /* Receive information from the conversation partner.             */
  /*-----------------------------------------------------------------*/
  'CMRCV conversation_ID receive_buffer requested_length',
        'data_received received_length status_received',
        'request_to_send_received return_code'
  say; say 'Routine called:  CMRCV'
  select
    when (return_code = CM_OK) then
      do
        call TraceParms 'data_received status_received',
        if (data_received ¬= CM_NO_DATA_RECEIVED) then
          do
            receive_buffer = left(receive_buffer,received_length)
            complete_line = complete_line || receive_buffer
          end
        if (data_received = CM_COMPLETE_DATA_RECEIVED) then
          do
            /*-------------------------------------------------------*/
            /* Use EXECIO to write the data to OUTPUT LOGFILE A      */
            /* and reset the complete_line variable to nulls.        */
            /*-------------------------------------------------------*/
            address command 'EXECIO 1 DISKW OUTPUT LOGFILE A (FINIS',
                            'STRING' complete_line
```

```
            complete_line = ''
          end
        /*-------------------------------------------------------------*/
        /* Determine whether a confirmation request has been           */
        /* received.  If so, respond with a positive reply.            */
        /*-------------------------------------------------------------*/
        if (status_received = CM_CONFIRM_RECEIVED) |,
           (status_received = CM_CONFIRM_SEND_RECEIVED) |,
           (status_received = CM_CONFIRM_DEALLOC_RECEIVED) then
          do
            /*---------------------------------------------------------*/
            /* Issue Confirmed to reply to the partner.                */
            /*---------------------------------------------------------*/
            'CMCFMD conversation_ID return_code'
            say; say 'Routine called:  CMCFMD'
            if (return_code ¬= CM_OK) then call ErrorHandler 'CMCFMD'
            call TraceParms 'conversation_ID return_code'
          end
      end
    otherwise
      call ErrorHandler 'CMRCV'
  end
end
/*-------------------------------------------------------------------*/
/* Deallocate the conversation normally.                             */
/*-------------------------------------------------------------------*/
'CMDEAL conversation_ID return_code'
say; say 'Routine called:  CMDEAL'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMDEAL'
call TraceParms

GetOut:
  exit

/*------------------------- Subroutines --------------------------*/
```

⋮

The server's SENDBACK EXEC now contains these lines:

```
/*===================================================================*/
/*  SENDBACK EXEC - Sample server application.                       */
/*===================================================================*/
```

⋮

```
/*-------------------------------------------------------------------*/
/* Start a Receive loop.                                             */
/* Receive data, status, or both from conversation partner.          */
/*-------------------------------------------------------------------*/
requested_file = ''
requested_length = 20
do until (CMRCV_return_code ¬= CM_OK) |,
         (status_received = CM_CONFIRM_DEALLOC_RECEIVED)
  'CMRCV conversation_ID receive_buffer requested_length',
        'data_received received_length status_received',
        'request_to_send_received return_code'
  CMRCV_return_code = return_code
  say; say 'Routine called:  CMRCV'
  select
    when (CMRCV_return_code = CM_OK) then
      do
        call TraceParms 'data_received status_received'
        if (data_received ¬= CM_NO_DATA_RECEIVED) the
          do
            receive_buffer = left(receive_buffer,received_length)
            requested_file = requested_file || receive_buffer
          end
```

```
        /*---------------------------------------------------------*/
        /* Determine whether a confirmation request has been       */
        /* received.  If so, respond with a positive reply.        */
        /*---------------------------------------------------------*/
        if (status_received = CM_CONFIRM_RECEIVED) |,
           (status_received = CM_CONFIRM_SEND_RECEIVED) |,
           (status_received = CM_CONFIRM_DEALLOC_RECEIVED) then
          do
            /*-----------------------------------------------------*/
            /* Issue Confirmed to reply to the partner.            */
            /*-----------------------------------------------------*/
            'CMCFMD conversation_ID return_code'
            say; say 'Routine called:  CMCFMD'
            if (return_code ¬= CM_OK) then call ErrorHandler 'CMCFMD'
            call TraceParms 'conversation_ID return_code'
          end
        if (status_received = CM_SEND_RECEIVED) |,
           (status_received = CM_CONFIRM_SEND_RECEIVED) then
          call SendFile
        else
          if (status_received = CM_CONFIRM_DEALLOC_RECEIVED) then
            do
              say; say 'Conversation deallocated by partner'
            end
      end
    when (CMRCV_return_code = CM_DEALLOCATED_NORMAL) then
      do
        call TraceParms 'data_received status_received'
        say; say 'Conversation deallocated by partner'
      end
    otherwise
      call ErrorHandler 'CMRCV'
  end
end

GetOut:
  exit

/*------------------------- Subroutines --------------------------*/
```

⋮

After filing both execs, start them up again by entering:

```
process getfile
```

and choose confirmation processing when prompted.

Here are the results displayed from the REQUESTR user ID's side of the conversation:

```
process getfile
Requesting the file:  TEST FILE A

Routine called:  CMINIT
  conversation_state is => CM_INITIALIZE_STATE

Would you like confirmation processing? (Y/N)
Y

Routine called:  CMSSL
  conversation_state is => CM_INITIALIZE_STATE
  Confirmation processing enabled

Routine called:  CMALLC
  conversation_state is => CM_SEND_STATE

Routine called:  CMSEND
  conversation_state is => CM_SEND_STATE

Routine called:  CMCFM
  conversation_ID is 00000000
  request_to_send_received is CM_REQ_TO_SEND_NOT_RECEIVED
  return_code is CM_OK
  conversation_state is => CM_SEND_STATE

Routine called:  CMPTR
  conversation_state is => CM_RECEIVE_STATE

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED
  conversation_state is => CM_RECEIVE_STATE

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED
  conversation_state is => CM_RECEIVE_STATE

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_SEND_RECEIVED
  conversation_state is => CM_SEND_STATE

Routine called:  CMDEAL
Ready;
```

Figure 36. Results of Confirmation Processing by PROCESS EXEC

And, here are the results displayed at the SERVR user ID:

```
Routine called:  CMACCP
  conversation_state is => CM_RECEIVE_STATE

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED
  conversation_state is => CM_RECEIVE_STATE

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_CONFIRM_RECEIVED
  conversation_state is => CM_CONFIRM_STATE

Routine called:  CMCFMD
  conversation_ID is 00000000
  return_code is CM_OK
  conversation_state is => CM_RECEIVE_STATE

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_CONFIRM_SEND_RECEIVED
  conversation_state is => CM_CONFIRM_SEND_STATE

Routine called:  CMCFMD
  conversation_ID is 00000000
  return_code is CM_OK
  conversation_state is => CM_SEND_STATE

Routine called:  CMSEND
  conversation_state is => CM_SEND_STATE

Routine called:  CMSEND
  conversation_state is => CM_SEND_STATE

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_CONFIRM_DEALLOC_RECEIVED
  conversation_state is => CM_CONFIRM_DEALLOCATE_STATE

Routine called:  CMCFMD
conversation_ID is 00000000
return_code is CM_OK

Conversation deallocated by partner
Ready;
```

Figure 37. Results of Confirmation Processing by SENDBACK EXEC

Our program still has the same basic flow to it, but there are some differences this time. Notice that whenever a confirmation request was made, it was reflected to the partner in the *status_received* parameter of a Receive (CMRCV) call.

Also notice the states that the conversation entered following receipt of a confirmation request. The conversation will remain in the **Confirm**, **Confirm-Send**, or **Confirm-Deallocate** state until the local program replies to the partner with Confirmed (CMCFMD), Send_Error (CMSERR), or Deallocate (CMDEAL) with *deallocate_type* set to CM_DEALLOCATE_ABEND.

FYI: Tidying Up, Part II

Now that we have seen the conversation state changes for several program executions, let's comment out that section of the TraceParms subroutine in both the PROCESS EXEC and the SENDBACK EXEC. Here is an easy way to make that change:

```
TraceParms:
/*-------------------------------------------------------------------*/
/* Display parameters and their values as passed to this subroutine.*/
/*-------------------------------------------------------------------*/
   ⋮
/*-------------------------------------------------------------------*/
/* Extract the current conversation state of the local program.     */
/*-------------------------------------------------------------------*/
/* Commenting out next four lines ...
'CMECS conversation_ID conversation_state return_code'
if (return_code = CM_OK) then
  say '  conversation_state is =>',
      cm_conversation_state.conversation_state
... */

return
```

Most of the previously added calls to TraceParms no longer pass a list of parameters, so those existing calls will not be providing any function now. If you choose to remove the calls or comment them out, go right ahead, but leaving them in the programs the way they are is also fine. We still want to keep the TraceParms calls in the Receive routine sections that pass the *data_received* and *status_received* values.

We will continue calling TraceParms once each time a new routine gets added to one of our programs to display the resulting parameters.

We have mentioned the *prepare_to_receive_type* and *deallocate_type* characteristics in the context of the last few communications routines, but we have not described their function. Now, let's take a closer look at what they are used for by discussing the routines that can set them.

The Set_Prepare_To_Receive_Type (CMSPTR) Call

The Set_Prepare_To_Receive_Type (CMSPTR) call sets the *prepare_to_receive_type* conversation characteristic for a given conversation and overrides the value assigned by the Initialize_Conversation (CMINIT) or Accept_Conversation (CMACCP) call.

The format for Set_Prepare_To_Receive_Type is:

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

Input Parameters

Use the ***conversation_ID*** parameter to identify the conversation.

Use the ***prepare_to_receive_type*** parameter to specify the type of prepare-to-receive processing to be performed for this conversation. You can set the *prepare_to_receive_type* variable to one of the following values:

CM_PREP_TO_RECEIVE_SYNC_LEVEL (0)
: Perform the prepare-to-receive based on one of the following *sync_level* settings:
  - If *sync_level* is CM_NONE, execute the function of the Flush (CMFLUS) call and enter **Receive** state.
  - If *sync_level* is CM_CONFIRM, execute the function of the Confirm (CMCFM) call and if successful (as indicated by a return code of CM_OK on the Prepare_To_Receive call, or a return code of CM_OK on the Send_Data call with *send_type* set to CM_SEND_AND_PREP_TO_RECEIVE), enter **Receive** state. If Confirm is not successful, the state of the conversation is determined by the return code.
  - If *sync_level* is CM_SYNC_POINT, enter **Defer-Receive** state until the program issues a synchronization point service's commit or backout call, or until the program issues a Confirm or Flush call for this conversation. If one of those calls is successful, enter **Receive** state. Otherwise, the conversation state is determined by the return code.

CM_PREP_TO_RECEIVE_FLUSH (1)
: Execute the function of the Flush (CMFLUS) call and enter **Receive** state.

CM_PREP_TO_RECEIVE_CONFIRM (2)
: Execute the function of the Confirm call and if successful (as indicated by a return code of CM_OK on the Prepare_To_Receive call, or a return code of CM_OK on the Send_Data call with *send_type* set to CM_SEND_AND_PREP_TO_RECEIVE), enter **Receive** state. If it is not successful, the state of the conversation is determined by the return code.

Output Parameter

The possible values for the ***return_code*** parameter are:

CM_OK (0)
: indicates that the *prepare_to_receive_type* has been set.

CM_PRODUCT_SPECIFIC_ERROR (20)
: indicates an error from CMS; check the CPICOMM LOGDATA file.

CM_PROGRAM_PARAMETER_CHECK (24)
: indicates that the specified conversation ID is unassigned, that the *prepare_to_receive_type* is CM_PREP_TO_RECEIVE_CONFIRM, but the conversation is assigned with *sync_level* set to CM_NONE, or that the *prepare_to_receive_type* is set to an undefined value.

Results of the Call

Anything other than a return code of CM_OK results in no change to the *prepare_to_receive_type* characteristic. This call does not cause a state change.

Adding CMSPTR to Our Requester Program

When *sync_level* is CM_CONFIRM, a program that sets the *prepare_to_receive_type* to CM_PREP_TO_RECEIVE_CONFIRM will behave exactly as a program that keeps the default value of CM_PREP_TO_RECEIVE_SYNC_LEVEL.

We saw the confirmation request that was received by the server following the requester's call to Prepare_To_Receive (CMPTR) the last time we ran our programs. Now let's set the *prepare_to_receive_type* to CM_PREP_TO_RECEIVE_FLUSH. This will have the same effect as following the Prepare_To_Receive call with a call to Flush (CMFLUS).

We will add the Set_Prepare_To_Receive call to the PROCESS EXEC just to see the difference in processing. Instead of the partner receiving a *status_received* value of CM_CONFIRM_SEND_RECEIVED, it should get CM_SEND_RECEIVED.

Let's add the call immediately before the Prepare_To_Receive call. Your exec should now have the following lines in it.

```
/*====================================================================*/
/*  PROCESS EXEC - Sample file requester application.                 */
/*====================================================================*/

   ⋮
/*--------------------------------------------------------------------*/
/* Confirm that partner has started and received the name of          */
/* the requested file.                                                */
/*--------------------------------------------------------------------*/
'CMCFM conversation_ID request_to_send_received',
      'return_code'
say; say 'Routine called:  CMCFM'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMCFM'
call TraceParms
/*--------------------------------------------------------------------*/
/* Set the prepare_to_receive_type to CM_PREP_TO_RECEIVE_FLUSH.       */
/*--------------------------------------------------------------------*/
prepare_to_receive_type = CM_PREP_TO_RECEIVE_FLUSH
'CMSPTR conversation_ID prepare_to_receive_type return_code'
say; say 'Routine called:  CMSPTR'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMSPTR'
call TraceParms 'conversation_ID prepare_to_receive_type return_code'
/*--------------------------------------------------------------------*/
/* Issue Prepare_To_Receive to switch the conversation state from     */
/* Send state to Receive state.                                       */
/*--------------------------------------------------------------------*/
'CMPTR conversation_ID return_code'
say; say 'Routine called:  CMPTR'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMPTR'
call TraceParms
   ⋮

/*-------------------------- Subroutines ---------------------------*/

TraceParms:
/*--------------------------------------------------------------------*/
/* Display parameters and their values as passed to this subroutine.*/
/*--------------------------------------------------------------------*/
parse arg parmlist
do word_num = 1 to words(parmlist)
  parameter = word(parmlist,word_num)
  select
    when (parameter = 'return_code') then
      say '  return_code is' cm_return_code.return_code
    when (parameter = 'buffer') then
      say '  buffer is' left(buffer,send_length)
    when (parameter = 'receive_buffer') then
      say '  buffer is' left(receive_buffer,received_length)
    when (parameter = 'data_received') then
      say '  data_received is' cm_data_received.data_received
    when (parameter = 'status_received') then
      say '  status_received is' cm_status_received.status_received
    when (parameter = 'request_to_send_received') then
      say '  request_to_send_received is',
          cm_request_to_send_received.request_to_send_received
    when (parameter = 'sync_level') then
      say '  sync_level is' cm_sync_level.sync_level
    when (parameter = 'prepare_to_receive_type') then
      say '  prepare_to_receive_type is',
```

```
          cm_prepare_to_receive_type.prepare_to_receive_type
    otherwise
      say ' ' parameter 'is' value(parameter)
  end
end
/*--------------------------------------------------------------------*/
/* Extract the current conversation state of the local program.       */
/*--------------------------------------------------------------------*/
/* Commenting out next four lines ...
'CMECS conversation_ID conversation_state return_code'
if (return_code = CM_OK) then
  say '  conversation_state is =>',
      cm_conversation_state.conversation_state
... */

return
```

⋮

File the exec and let's try it out with

```
process getfile
```

and again pick confirmation processing.

The terminal session from the REQUESTR virtual machine is:

```
process getfile
Requesting the file:  TEST FILE A

Routine called:  CMINIT

Would you like confirmation processing? (Y/N)
Y

Routine called:  CMSSL
  Confirmation processing enabled

Routine called:  CMALLC

Routine called:  CMSEND

Routine called:  CMCFM

Routine called:  CMSPTR
  conversation_ID is 00000000
  prepare_to_receive_type is CM_PREP_TO_RECEIVE_FLUSH
  return_code is CM_OK

Routine called:  CMPTR

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_SEND_RECEIVED

Routine called:  CMDEAL
Ready;
```

Figure 38. Results after Adding CMSPTR Call to PROCESS EXEC

And at the SERVR virtual machine, you will see:

```
Routine called:  CMACCP

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_CONFIRM_RECEIVED

Routine called:  CMCFMD

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_SEND_RECEIVED

Routine called:  CMSEND

Routine called:  CMSEND

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_CONFIRM_DEALLOC_RECEIVED

Routine called:  CMCFMD

Conversation deallocated by partner
Ready;
```

Figure 39. Results of SENDBACK EXEC Execution

This time the server's Receive (CMRCV) call, which completes following the requester's Prepare_To_Receive call, returned a *status_received* value of CM_SEND_RECEIVED. Before we changed the *prepare_to_receive_type*, that same Receive completed with CM_CONFIRM_SEND_RECEIVED in *status_received*.

Keep in mind that the *prepare_to_receive_type* for the requester's end of the conversation will continue to be CM_PREP_TO_RECEIVE_FLUSH for the rest of the conversation, unless a subsequent call to Set_Prepare_To_Receive_Type resets it.

You also should understand that the *prepare_to_receive_type* for the server's end of the conversation remains unchanged at this point. It is still set to the default value CM_PREP_TO_RECEIVE_SYNC_LEVEL.

The Set_Send_Type (CMSST) Call

The Set_Send_Type (CMSST) call sets the *send_type* characteristic for a given conversation and overrides the value that was assigned with the Initialize_Conversation (CMINIT) or Accept_Conversation (CMALLC) call.

The format for Set_Send_Type is:

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

Input Parameters

Use the ***conversation_ID*** parameter to identify the conversation.

Use the ***send_type*** parameter to specify what information, if any, is to be sent to the remote program in addition to the data supplied on the Send_Data (CMSEND) call, and whether the data is to be sent immediately or buffered. You can set the *send_type* variable to one of the following values:

CM_BUFFER_DATA (0)
: No additional data is sent; supplied data could be buffered.

CM_SEND_AND_FLUSH (1)
: No additional data is sent; supplied data is sent immediately.

CM_SEND_AND_CONFIRM (2)
: Supplied data is sent immediately along with a confirmation request.

CM_SEND_AND_PREP_TO_RECEIVE (3)
: Supplied data is sent immediately along with send control of the conversation.

CM_SEND_AND_DEALLOCATE (4)
: Supplied data is sent immediately along with deallocation notification.

Output Parameter

The possible ***return_code*** values are:

CM_OK (0)
: indicates that the *send_type* value has been set.

CM_PRODUCT_SPECIFIC_ERROR (20)
: indicates a CMS error; check the CPICOMM LOGDATA file.

CM_PROGRAM_PARAMETER_CHECK (24)
: indicates that the specified conversation ID is unassigned, that an undefined value was specified for *send_type*, or that there is a conflict between the *send_type* value and the *sync_level* value.

Results of the Call

Anything other than a return code of CM_OK results in no change to the *send_type* characteristic. This call does not cause a state change.

Adding CMSST to Our Server Program

Let's add the Set_Send_Type call to the SENDBACK EXEC on the SERVR user ID. We will use a *send_type* value of CM_SEND_AND_PREP_TO_RECEIVE.

That change will affect the behavior of any Send_Data calls that follow it because a state change notification will be sent along with the file contents. However, we do not want to transfer send control of the conversation to the file requester partner until after the entire file has been sent. For that reason, the Set_Send_Type call needs to be inserted into the Send_Data loop so that the *send_type* is set just before the last Send_Data call is issued.

Your exec should now have the following lines:

```
/*====================================================================*/
/*  SENDBACK EXEC - Sample server application.                        */
/*====================================================================*/

  ⋮

/*------------------------- Subroutines --------------------------*/

SendFile:
/*--------------------------------------------------------------------*/
/* Read the contents of the requested file and send each line of    */
/* the file to the partner program.                                   */
/*--------------------------------------------------------------------*/
address command 'EXECIO * DISKR' requested_file '(FINIS STEM LINE.'
do index = 1 to line.0
```

```
  if (index = line.0) then
    /*-----------------------------------------------------------------*/
    /* Reset the send_type conversation characteristic just         */
    /* before the final Send_Data call.                             */
    /*-----------------------------------------------------------------*/
    do
      send_type = CM_SEND_AND_PREP_TO_RECEIVE
      'CMSST conversation_ID send_type return_code'
      say; say 'Routine called:  CMSST'
      if (return_code ¬= CM_OK) then call ErrorHandler 'CMSST'
      call TraceParms 'conversation_ID send_type return_code'
    end
  buffer = line.index
  send_length = length(buffer)
  'CMSEND conversation_ID buffer send_length',
         'request_to_send_received return_code'
  say; say 'Routine called:  CMSEND'
  if (return_code ¬= CM_OK) then call ErrorHandler 'CMSEND'
  call TraceParms
end

return


TraceParms:
/*-------------------------------------------------------------------*/
/* Display parameters and their values as passed to this subroutine.*/
/*-------------------------------------------------------------------*/
parse arg parmlist
do word_num = 1 to words(parmlist)
  parameter = word(parmlist,word_num)
  select
    when (parameter = 'return_code') then
      say '  return_code is' cm_return_code.return_code
    when (parameter = 'buffer') then
      say '  buffer is' left(buffer,send_length)
    when (parameter = 'receive_buffer') then
      say '  buffer is' left(receive_buffer,received_length)
    when (parameter = 'data_received') then
      say '  data_received is' cm_data_received.data_received
    when (parameter = 'status_received') then
      say '  status_received is' cm_status_received.status_received
    when (parameter = 'request_to_send_received') then
      say '  request_to_send_received is',
          cm_request_to_send_received.request_to_send_received
    when (parameter = 'send_type') then
      say '  send_type is' cm_send_type.send_type
    otherwise
      say ' ' parameter 'is' value(parameter)
  end
end
/*-------------------------------------------------------------------*/
/* Extract the current conversation state of the local program.     */
/*-------------------------------------------------------------------*/
/* Commenting out next four lines ...
'CMECS conversation_ID conversation_state return_code'
if (return_code = CM_OK) then
  say '  conversation_state is =>',
      cm_conversation_state.conversation_state
... */

return
```

⋮

Now file the exec, and let's execute it. Start it with

```
process getfile
```

and choose confirmation processing.

The results on the requester side should be:

```
process getfile
Requesting the file:  TEST FILE A

Routine called:  CMINIT

Would you like confirmation processing? (Y/N)
Y

Routine called:  CMSSL
  Confirmation processing enabled

Routine called:  CMALLC

Routine called:  CMSEND

Routine called:  CMCFM

Routine called:  CMSPTR

Routine called:  CMPTR

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_CONFIRM_SEND_RECEIVED

Routine called:  CMCFMD

Routine called:  CMDEAL
Ready;
```

Figure 40. Results of PROCESS EXEC Execution

The server side's results should be:

```
Routine called:  CMACCP

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_CONFIRM_RECEIVED

Routine called:  CMCFMD

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_SEND_RECEIVED

Routine called:  CMSEND

Routine called:  CMSST
  conversation_ID is 00000000
  send_type is CM_SEND_AND_PREP_TO_RECEIVE
  return_code is CM_OK

Routine called:  CMSEND

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_CONFIRM_DEALLOC_RECEIVED

Routine called:  CMCFMD

Conversation deallocated by partner
Ready;
```

Figure 41. Results after Adding CMSST Call to SENDBACK EXEC

The results are basically the same as the last time we executed our programs. However, the last Receive call on the REQUESTR side of the conversation indicates that the *status_received* value is CM_CONFIRM_SEND_RECEIVED this time, rather than CM_SEND_RECEIVED. Do you understand what happened?

By the time the final Send_Data call is issued by the server, the program has reset the *send_type* to CM_SEND_AND_PREP_TO_RECEIVE. When the Send_Data routine is called, that *send_type* value causes send control of the conversation to be sent to the partner program along with any data that is supplied. The same results can be achieved by issuing Send_Data with the default *send_type* of CM_BUFFER_DATA followed by a Prepare_To_Receive call.

We need to remember, though, that the *prepare_to_receive_type* characteristic also comes into the picture because of that implicit Prepare_To_Receive call. The Set_Prepare_To_Receive_Type (CMSPTR) call that the PROCESS EXEC issues affects only the file requester's end of the conversation. So, the server's *prepare_to_receive_type* has not changed from the default of CM_PREP_TO_RECEIVE_SYNC_LEVEL.

Now, the *sync_level* of the conversation must be taken into consideration. Because we request confirmation processing, the *sync_level* characteristic is set to CM_CONFIRM. When the *prepare_to_receive_type* is CM_PREP_TO_RECEIVE_SYNC_LEVEL and the *sync_level* is CM_CONFIRM, an implicit confirmation request is sent to the partner. If a positive response is received, the local side of the conversation enters **Receive** state.

The Receive (CMRCV) call following the Send_Data call in the SENDBACK EXEC had been changing the conversation state. Now, that state change is happening as a result of the Send_Data call, instead.

In effect, when the Send_Data call is made, it is as if our program also issued a Prepare_To_Receive (CMPTR) call and a Confirm (CMCFM) call. The point behind our example is twofold. First, as we have already seen, it is possible to combine the function of multiple calls into a single call. In some cases, this capability may simplify your programs. Second, you need to keep this fact in mind so that you can write your programs to correctly anticipate this type of behavior.

The Set_Deallocate_Type (CMSDT) Call

The Set_Deallocate_Type (CMSDT) call sets the *deallocate_type* characteristic for a given conversation and overrides the value assigned with either the Initialize_Conversation (CMINIT) or Accept_Conversation (CMACCP) call.

The format for Set_Deallocate_Type is:

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

Input Parameters

Use the ***conversation_ID*** parameter to identify the conversation.

Use the ***deallocate_type*** parameter to specify the type of deallocation to be performed. You can set it to one of the following values:

CM_DEALLOCATE_SYNC_LEVEL (0)
: perform deallocation based on the *sync_level* characteristic in effect for this conversation:
  - If *sync_level* is CM_NONE, execute the function of the Flush (CMFLUS) call and deallocate the conversation normally and unconditionally.
  - If *sync_level* is CM_CONFIRM, execute the function of the Confirm (CMCFM) call. The conversation is deallocated normally when the remote program responds to the confirmation request by issuing the Confirmed (CMCFMD) call. The conversation remains allocated when the remote program responds to the confirmation request by issuing the Send_Error (CMSERR) call.
  - If *sync_level* is CM_SYNC_POINT, defer the deallocation until the program issues a synchronization point service's commit call. If the commit call is successful, the conversation is deallocated normally. If the commit is not successful or if the program issues a synchronization point service's backout call instead of a commit, the conversation is not deallocated.

CM_DEALLOCATE_FLUSH (1)
: execute the function of the Flush call and deallocate the conversation normally.

CM_DEALLOCATE_CONFIRM (2)
: execute the function of the Confirm call. The conversation is deallocated normally when the remote program responds to the confirmation request by issuing the Confirmed call. The conversation remains allocated if the remote program responds to the confirmation request by issuing the Send_Error (CMSERR) call.

CM_DEALLOCATE_ABEND (3)
: execute the function of the Flush call when the program is in **Send** state

and deallocate the conversation abnormally. If the program is in **Receive** state, data purging can occur. This *deallocate_type* is used to unconditionally deallocate the conversation regardless of the level of synchronization, and is intended for use when a program detects an error condition that prevents further useful communications.

Output Parameter

The possible values for the ***return_code*** parameter are:

CM_OK (0)
: indicates that the *deallocate_type* value has been set.

CM_PRODUCT_SPECIFIC_ERROR (20)
: indicates a CMS error; check the CPICOMM LOGDATA file.

CM_PROGRAM_PARAMETER_CHECK (24)
: indicates that the specified conversation ID is unassigned, that there is a conflict between the *sync_level* and the *deallocate_type* values, or that the *deallocate_type* specifies an undefined value.

Results of the Call

When *return_code* is anything other than CM_OK, the *deallocate_type* characteristic is unchanged. This call does not cause a state change.

Adding CMSDT to Both Our Programs

Setting the *deallocate_type* to CM_DEALLOCATE_FLUSH will make a Deallocate (CMDEAL) call act like one with a *deallocate_type* of CM_DEALLOCATE_SYNC_LEVEL combined with a *sync_level* of CM_NONE. These are the default values for these conversation characteristics.

A Deallocate with *deallocate_type* set to CM_DEALLOCATE_CONFIRM is the same as one with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL combined with *sync_level* set to CM_CONFIRM. That is how our program is currently working.

The *deallocate_type* that is a little more interesting is CM_DEALLOCATE_ABEND, which a program would use when it determines there is a problem that will prevent further communications with its partner.

When our programs detect a bad *return_code* value, they call an error routine and exit. The best way to leave any conversation is to first issue Deallocate, and now we have a way of successfully calling Deallocate regardless of the conversation's current state.

Let's change the error subroutines Error and ErrorHandler in both programs to call a new subroutine named AbnormalEnd. The AbnormalEnd routine will issue a Set_Deallocate_Type call with *deallocate_type* set to CM_DEALLOCATE_ABEND, followed by a Deallocate call. The required update is similar in both execs.

The PROCESS exec should include the following additions:

```
/*================================================================*/
/*  PROCESS EXEC - Sample file requester application.             */
/*================================================================*/

  ⋮

/*------------------------- Subroutines --------------------------*/

TraceParms:
/*----------------------------------------------------------------*/
/* Display parameters and their values as passed to this subroutine.*/
/*----------------------------------------------------------------*/
```

```
parse arg parmlist
do word_num = 1 to words(parmlist)
  parameter = word(parmlist,word_num)
  select
    when (parameter = 'return_code') then
      say '  return_code is' cm_return_code.return_code
    when (parameter = 'buffer') then
      say '  buffer is' left(buffer,send_length)
    when (parameter = 'receive_buffer') then
      say '  buffer is' left(receive_buffer,received_length)
    when (parameter = 'data_received') then
      say '  data_received is' cm_data_received.data_received
    when (parameter = 'status_received') then
      say '  status_received is' cm_status_received.status_received
    when (parameter = 'request_to_send_received') then
      say '  request_to_send_received is',
          cm_request_to_send_received.request_to_send_received
    when (parameter = 'sync_level') then
      say '  sync_level is' cm_sync_level.sync_level
    when (parameter = 'prepare_to_receive_type') then
      say '  prepare_to_receive_type is',
          cm_prepare_to_receive_type.prepare_to_receive_type
    when (parameter = 'deallocate_type') then
      say '  deallocate_type is' cm_deallocate_type.deallocate_type
    otherwise
      say ' ' parameter 'is' value(parameter)
  end
end
/*--------------------------------------------------------------------*/
/* Extract the current conversation state of the local program.       */
/*--------------------------------------------------------------------*/
/* Commenting out next four lines ...
'CMECS conversation_ID conversation_state return_code'
if (return_code = CM_OK) then
  say '  conversation_state is =>',
      cm_conversation_state.conversation_state
... */

return


Error:
/*--------------------------------------------------------------------*/
/* Report error when REXX special variable RC is not 0.               */
/*--------------------------------------------------------------------*/
say
say '* ERROR:  REXX has detected an error'
say '          The return code variable RC was set to' rc
call AbnormalEnd
signal GetOut


ErrorHandler:
/*--------------------------------------------------------------------*/
/* Report routine that failed and the error return code.              */
/*--------------------------------------------------------------------*/
parse arg routine_name
say
say '* ERROR:  An error occurred during a' routine_name 'call'
say '          The return_code was set to' cm_return_code.return_code
call AbnormalEnd
signal GetOut


AbnormalEnd:
/*--------------------------------------------------------------------*/
/* Abnormally deallocate the conversation.  Since we are exiting      */
```

```
/* due to an error, we will not display an error message if the     */
/* Set_Deallocate_Type or Deallocate call encounters an error.      */
/*------------------------------------------------------------------*/
deallocate_type = CM_DEALLOCATE_ABEND
'CMSDT conversation_ID deallocate_type return_code'
say; say 'Routine called:  CMSDT'
if (return_code = CM_OK) then
  do
    call TraceParms 'conversation_ID deallocate_type return_code'
    'CMDEAL conversation_ID return_code'
    say; say 'Routine called:  CMDEAL'
    if (return_code = CM_OK) then
      call TraceParms 'conversation_ID return_code'
  end

return
```

And, the SENDBACK exec should include these additional lines:

```
/*==================================================================*/
/*  SENDBACK EXEC - Sample server application.                      */
/*==================================================================*/

   ⋮

/*------------------------- Subroutines --------------------------*/

   ⋮

TraceParms:
/*------------------------------------------------------------------*/
/* Display parameters and their values as passed to this subroutine.*/
/*------------------------------------------------------------------*/
parse arg parmlist
do word_num = 1 to words(parmlist)
  parameter = word(parmlist,word_num)
  select
    when (parameter = 'return_code') then
      say '  return_code is' cm_return_code.return_code
    when (parameter = 'buffer') then
      say '  buffer is' left(buffer,send_length)
    when (parameter = 'receive_buffer') then
      say '  buffer is' left(receive_buffer,received_length)
    when (parameter = 'data_received') then
      say '  data_received is' cm_data_received.data_received
    when (parameter = 'status_received') then
      say '  status_received is' cm_status_received.status_received
    when (parameter = 'request_to_send_received') then
      say '  request_to_send_received is',
          cm_request_to_send_received.request_to_send_received
    when (parameter = 'send_type') then
      say '  send_type is' cm_send_type.send_type
    when (parameter = 'deallocate_type') then
      say '  deallocate_type is' cm_deallocate_type.deallocate_type
    otherwise
      say ' ' parameter 'is' value(parameter)
  end
end
/*------------------------------------------------------------------*/
/* Extract the current conversation state of the local program.     */
/*------------------------------------------------------------------*/
/* Commenting out next four lines ...
'CMECS conversation_ID conversation_state return_code'
if (return_code = CM_OK) then
  say '  conversation_state is =>',
      cm_conversation_state.conversation_state
... */
```

```
return


Error:
/*--------------------------------------------------------------------*/
/* Report error when REXX special variable RC is not 0.               */
/*--------------------------------------------------------------------*/
say
say '* ERROR:  REXX has detected an error'
say '          The return code variable RC was set to' rc
call AbnormalEnd
signal GetOut


ErrorHandler:
/*--------------------------------------------------------------------*/
/* Report routine that failed and the error return code.              */
/*--------------------------------------------------------------------*/
parse arg routine_name
say
say '* ERROR:  An error occurred during a' routine_name 'call'
say '          The return_code was set to' cm_return_code.return_code
call AbnormalEnd
signal GetOut


AbnormalEnd:
/*--------------------------------------------------------------------*/
/* Abnormally deallocate the conversation.  Since we are exiting      */
/* due to an error, we will not display an error message if the       */
/* Set_Deallocate_Type or Deallocate call encounters an error.        */
/*--------------------------------------------------------------------*/
deallocate_type = CM_DEALLOCATE_ABEND
'CMSDT conversation_ID deallocate_type return_code'
say; say 'Routine called:  CMSDT'
if (return_code = CM_OK) then
  do
    call TraceParms 'conversation_ID deallocate_type return_code'
    'CMDEAL conversation_ID return_code'
    say; say 'Routine called:  CMDEAL'
    if (return_code = CM_OK) then
      call TraceParms 'conversation_ID return_code'
  end

return
```

Now file the execs and enter

```
process getfile
```

but **do not** select confirmation processing this time.

The results on the REQUESTR user ID should be:

```
process getfile
Requesting the file:  TEST FILE A

Routine called:  CMINIT

Would you like confirmation processing? (Y/N)
N

Routine called:  CMALLC

Routine called:  CMSEND

Routine called:  CMCFM

* ERROR:  An error occurred during a CMCFM call
          The return_code was set to CM_PROGRAM_PARAMETER_CHECK

Routine called:  CMSDT
  conversation_ID is 00000000
  deallocate_type is CM_DEALLOCATE_ABEND
  return_code is CM_OK

Routine called:  CMDEAL
  conversation_ID is 00000000
  return_code is CM_OK
Ready;
```

Figure 42. Results after Adding CMSDT Call to PROCESS EXEC

The results on the SERVR user ID should be:

```


Routine called:  CMACCP

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Routine called:  CMRCV

* ERROR:  An error occurred during a CMRCV call
          The return_code was set to CM_DEALLOCATED_ABEND

Routine called:  CMSDT
Ready;
```

Figure 43. Results after Adding CMSDT Call to SENDBACK EXEC

Let's start our analysis of what happened with the requester. Because we answered "no" to the confirmation prompt, Set_Sync_Level was not called to change the *sync_level* conversation characteristic. Therefore, when the requester program tried to issue a Confirm call, the *sync_level* characteristic's value was CM_NONE. This attempt caused the CM_PROGRAM_PARAMETER_CHECK *return_code*. The requester's ErrorHandler routine was called and Set_Deallocate_Type was called to set the *deallocate_type* to CM_DEALLOCATE_ABEND. Deallocate was then issued to terminate the conversation.

That abnormal termination was reflected to the server program on a Receive call that completed with a *return_code* of CM_DEALLOCATED_ABEND. When the server program detected that something was wrong, it called the error subroutine, which called Set_Deallocate_Type with *deallocate_type* set to

CM_DEALLOCATE_ABEND for its side of the conversation. But because the deallocation of the conversation by the requester had already been completed, the server's end of the conversation was in **Reset** state. Thus, the attempt to set the *deallocate_type* failed because the conversation ID was no longer assigned. (Recall that we do not bother to display an error message if the Set_Deallocate_Type call fails, because the conversation is already being deallocated because of an error.)

While trying out the CM_DEALLOCATE_ABEND *deallocate_type*, we also discovered a logic error in the requester application. These sample programs are handling conversations with *sync_level* set to either CM_NONE or CM_CONFIRM, so the requester should not call Confirm unless confirmation processing has been enabled.

Because the PROCESS EXEC sets the *sync_level* to CM_CONFIRM based on console input, we can add conditional logic preceding the Confirm call to ensure that Confirm is called only when confirmation processing is requested.

The change to the PROCESS EXEC would look like:

```
/*====================================================================*/
/*  PROCESS EXEC - Sample file requester application.                 */
/*====================================================================*/
   ⋮
/*--------------------------------------------------------------------*/
/* Send the name of the file being requested to the partner program.*/
/*--------------------------------------------------------------------*/
buffer = fname ftype fmode
send_length = length(buffer)
'CMSEND conversation_ID buffer send_length',
       'request_to_send_received return_code'
say; say 'Routine called:  CMSEND'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMSEND'
call TraceParms
/*--------------------------------------------------------------------*/
/* Call Confirm only when sync_level is not CM_NONE.  We can use      */
/* the confirmation processing flag set from console input.           */
/*--------------------------------------------------------------------*/
if (perform_confirm = 'Y') then
  do
    /*----------------------------------------------------------------*/
    /* Confirm that partner has started and received the name of      */
    /* the requested file.                                            */
    /*----------------------------------------------------------------*/
    'CMCFM conversation_ID request_to_send_received',
          'return_code'
    say; say 'Routine called:  CMCFM'
    if (return_code ¬= CM_OK) then call ErrorHandler 'CMCFM
    call TraceParms
  end
/*--------------------------------------------------------------------*/
/* Set the prepare_to_receive_type to CM_PREP_TO_RECEIVE_FLUSH.       */
/*--------------------------------------------------------------------*/
prepare_to_receive_type = CM_PREP_TO_RECEIVE_FLUSH
'CMSPTR conversation_ID prepare_to_receive_type return_code'
say; say 'Routine called:  CMSPTR'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMSPTR'
call TraceParms
```

⋮

The Extract_Conversation_Type (CMECT) Call

The Extract_Conversation_Type (CMECT) call extracts the value of the *conversation_type* characteristic for a given conversation. This routine is useful for a server application that wants to determine what the *conversation_type* is for a conversation it has accepted.

When the server accepts a conversation, the *conversation_type* characteristic has already been defined by the requester. Perhaps a server is written to send data with the expectation that the conversation is mapped, like our file-request server. With the addition of an Extract_Conversation_Type call immediately following an Accept_Conversation (CMACCP) call, the server can determine if the conversation it just accepted is mapped or basic. If the conversation is basic, the server can then call Set_Deallocate_Type (CMSDT) to set the *deallocate_type* to CM_DEALLOCATE_ABEND and issue a Deallocate (CMDEAL) call to terminate the conversation.

Alternatively, suppose the server wants to handle both mapped and basic conversations. A program issuing the Send_Data (CMSEND) call for a basic conversation must add a logical record length field as part of the *buffer* parameter. (This is done automatically by CPI Communications on a mapped conversation, which is one of the reasons we are using a mapped conversation between our two applications.) By issuing an Extract_Conversation_Type call, a server program can determine whether it needs to perform this type of extra processing. The *z/VM: CMS Application Development Guide* contains a sample CPI Communications resource manager program that uses the Extract_Conversation_Type call in a similar way.

The format for Extract_Conversation_Type is:

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

Input Parameter

Use the ***conversation_ID*** parameter to identify the conversation.

Output Parameters

The ***conversation_type*** parameter is a variable for returning the *conversation_type* characteristic of the specified conversation. Possible values it can return are:

CM_BASIC_CONVERSATION (0)
CM_MAPPED_CONVERSATION (1)

Possible values for the ***return_code*** parameter are:

CM_OK (0)
: indicates that the *conversation_type* has been extracted successfully.

CM_PRODUCT_SPECIFIC_ERROR (20)
: indicates a CMS error; check the CPICOMM LOGDATA file.

CM_PROGRAM_PARAMETER_CHECK (24)
: indicates that the specified conversation ID is unassigned.

Results of the Call

If the return code is not CM_OK, no *conversation_type* will be returned to the local program. The call neither changes the *conversation_type* for the specified conversation, nor causes a state change.

Adding CMECT to Our Server Program

Let's add the Extract_Conversation_Type call to the SENDBACK EXEC, following the Accept_Conversation (CMACCP) call.

If the conversation is basic, we will want to deallocate it abnormally. A Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND is already coded in the ErrorHandler subroutine, so let's just call it to terminate the conversation.

The updated exec should look like:

```
/*====================================================================*/
/*  SENDBACK EXEC - Sample server application.                        */
/*====================================================================*/

   ⋮
/*--------------------------------------------------------------------*/
/* Accept the incoming conversation.                                  */
/*--------------------------------------------------------------------*/
'CMACCP conversation_ID return_code'
say; say 'Routine called:  CMACCP'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMACCP'
call TraceParms
/*--------------------------------------------------------------------*/
/* Extract conversation_type to ensure the conversation is mapped.  */
/*--------------------------------------------------------------------*/
'CMECT conversation_ID conversation_type return_code'
say; say 'Routine called:  CMECT'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMECT'
call TraceParms 'conversation_ID conversation_type return_code'
/*--------------------------------------------------------------------*/
/* If the conversation is basic, deallocate abnormally.               */
/*--------------------------------------------------------------------*/
if (conversation_type = CM_BASIC_CONVERSATION) then
  do
    say; say '* ERROR:  Accepting and deallocating a basic',
             'conversation'
    call AbnormalEnd
    signal GetOut
  end
/*--------------------------------------------------------------------*/
/* Start a Receive loop.                                              */
/* Receive data, status, or both from conversation partner.           */
/*--------------------------------------------------------------------*/
requested_file = ''
requested_length = 20
   ⋮

/*------------------------- Subroutines --------------------------*/

   ⋮

TraceParms:
/*--------------------------------------------------------------------*/
/* Display parameters and their values as passed to this subroutine.*/
/*--------------------------------------------------------------------*/
parse arg parmlist
do word_num = 1 to words(parmlist)
  parameter = word(parmlist,word_num)
  select
```

```
    when (parameter = 'return_code') then
      say '  return_code is' cm_return_code.return_code
    when (parameter = 'buffer') then
      say '  buffer is' left(buffer,send_length)
    when (parameter = 'receive_buffer') then
      say '  buffer is' left(receive_buffer,received_length)
    when (parameter = 'data_received') then
      say '  data_received is' cm_data_received.data_received
    when (parameter = 'status_received') then
      say '  status_received is' cm_status_received.status_received
    when (parameter = 'request_to_send_received') then
      say '  request_to_send_received is',
          cm_request_to_send_received.request_to_send_received
    when (parameter = 'send_type') then
      say '  send_type is' cm_send_type.send_type
    when (parameter = 'deallocate_type') then
      say '  deallocate_type is' cm_deallocate_type.deallocate_type
    when (parameter = 'conversation_type') then
      say '  conversation_type is',
          cm_conversation_type.conversation_type
    otherwise
      say ' ' parameter 'is' value(parameter)
  end
end
/*--------------------------------------------------------------------*/
/* Extract the current conversation state of the local program.       */
/*--------------------------------------------------------------------*/
/* Commenting out next four lines ...
'CMECS conversation_ID conversation_state return_code'
if (return_code = CM_OK) then
  say '  conversation_state is =>',
      cm_conversation_state.conversation_state
... */

return
```

⋮

Before we try out the update we just made, however, let's look at a routine that can be used to alert the partner that an error may have occurred.

The Send_Error (CMSERR) Call

A program can use the Send_Error (CMSERR) call to inform its partner that it detected an error during a conversation. If the conversation is in **Send** state when Send_Error is issued, the call forces the LU to flush its send buffer.

Upon completion of a successful Send_Error call, the local program is in **Send** state and the remote program is in **Receive** state.

A program can use this routine to truncate an incomplete logical record it is sending, to inform the remote program of an error detected in received data, or to reject a confirmation request. In some situations, it may be useful to follow this call with a Send_Data (CMSEND) call to provide further information to the partner.

The format for Send_Error is:

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

Input Parameter

Use the ***conversation_ID*** parameter to identify the conversation.

Output Parameters

The ***request_to_send_received*** parameter is a variable for returning an indication of whether a request-to-send notification has been received from the partner program. It can return one of the following values:

CM_REQ_TO_SEND_NOT_RECEIVED (0)
CM_REQ_TO_SEND_RECEIVED (1)

If a request-to-send notification was received, it means that the remote program has requested that the local program's end of the conversation enter **Receive** state, which would place the remote program's end of the conversation in **Send** state.

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

Depending on the state of the conversation, some of the possible ***return_code*** values that can be returned are:

CM_OK (0)
: indicates that the Send_Error call executed successfully.

CM_DEALLOCATED_ABEND (17)
: usually indicates that the remote program deallocated the conversation with *deallocate_type* set to CM_DEALLOCATE_ABEND, or the remote LU did so because of a remote program abnormal-ending condition. The conversation is in **Reset** state.

CM_PRODUCT_SPECIFIC_ERROR (20)
: indicates a CMS error; check the CPICOMM LOGDATA file.

CM_PROGRAM_ERROR_PURGING (22)
: indicates that the remote program issued a Send_Error call and the conversation for the remote program was in **Receive** or **Confirm** state. The call may have caused information to be purged. Purging occurs when the remote program issues Send_Error for a conversation in **Receive** state before receiving all the information that the local program sent (all of the information sent before the CM_PROGRAM_ERROR_PURGING return code is reported to the local program).

CM_PROGRAM_PARAMETER_CHECK (24)
: indicates that the specified conversation ID is unassigned.

CM_PROGRAM_STATE_CHECK (25)
: indicates that the conversation is not in **Send**, **Receive**, **Send-Pending**, **Confirm**, **Confirm-Send**, **Confirm-Deallocate**, **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** state.

Allocation errors can also be returned on a Send_Error call.

Results of the Call

When *return_code* indicates CM_OK, the conversation enters **Send** state if the call is issued in **Receive**, **Confirm**, **Confirm-Send**, **Confirm-Deallocate**, or **Send-Pending** state. No state change occurs when the call is issued in **Send** state.

Adding CMSERR to Our Server Program

For our example, we will have SENDBACK EXEC call Send_Error if it detects that a basic conversation has been accepted. Let's add the call after the Extract_Conversation_Type (CMECT) call we just added, immediately before the ErrorHandler is called to deallocate the conversation.

The server program should include these changes:

```
/*====================================================================*/
/*  SENDBACK EXEC - Sample server application.                        */
/*====================================================================*/

   ⋮
/*--------------------------------------------------------------------*/
/* Extract conversation_type to ensure the conversation is mapped.  */
/*--------------------------------------------------------------------*/
'CMECT conversation_ID conversation_type return_code'
say; say 'Routine called:  CMECT'
if (return_code >= CM_OK) then call ErrorHandler 'CMECT'
call TraceParms 'conversation_ID conversation_type return_code'
/*--------------------------------------------------------------------*/
/* If the conversation is basic, deallocate abnormally.             */
/*--------------------------------------------------------------------*/
if (conversation_type = CM_BASIC_CONVERSATION) then
  do
    say; say '* ERROR:  Accepting and deallocating a basic',
             'conversation'
    /*----------------------------------------------------------------*/
    /* Call Send_Error to notify partner that error was detected.   */
    /* Since the program is going to exit, do not check the         */
    /* Send_Error results for an error.                             */
    /*----------------------------------------------------------------*/
    'CMSERR conversation_ID request_to_send_received return_code'
    say; say 'Routine called:  CMSERR'
    if (return_code = CM_OK) then
      call TraceParms 'conversation_ID request_to_send_received',
                      'return_code'
    call AbnormalEnd
    signal GetOut
  end
/*--------------------------------------------------------------------*/
/* Start a Receive loop.                                            */
/* Receive data, status, or both from conversation partner.         */
/*--------------------------------------------------------------------*/
requested_file = ''
requested_length = 20
   ⋮
```

Now, the only way to test the changes we have made is to have a basic conversation accepted by the server. To create a basic conversation, we need to issue the Set_Conversation_Type call.

The Set_Conversation_Type (CMSCT) Call

The Set_Conversation_Type (CMSCT) call sets the *conversation_type* characteristic for a given conversation, overriding the value assigned with the Initialize_Conversation (CMINIT) call.

Only the program initiating a conversation (using the Initialize_Conversation call) can issue the Set_Conversation_Type call. The call must be issued while in **Initialize** state, prior to the Allocate (CMALLC) call for the specified conversation.

The default *conversation_type* supplied by CPI Communications during conversation initialization is CM_MAPPED_CONVERSATION, so the only time a program would need to call Set_Conversation_Type is when a basic conversation is required.

The format for Set_Conversation_Type is:

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

Input Parameters

Use the ***conversation_ID*** parameter to identify the conversation.

Use the ***conversation_type*** parameter to specify the type of conversation to be allocated. You can set it to one of the following values:

CM_BASIC_CONVERSATION (0)
CM_MAPPED_CONVERSATION (1)

Output Parameter

Possible values for the ***return_code*** parameter are:

CM_OK (0)
: indicates that the *conversation_type* has been set.

CM_PRODUCT_SPECIFIC_ERROR(20)
: indicates a CMS error; check the CPICOMM LOGDATA file.

CM_PROGRAM_PARAMETER_CHECK (24)
: indicates that the specified conversation ID is unassigned, that the *conversation_type* specifies an undefined value, or that the *conversation_type* is set to CM_MAPPED_CONVERSATION, but the *fill* characteristic is set to CM_FILL_BUFFER or a prior call to Set_Log_Data is still in effect.

CM_PROGRAM_STATE_CHECK (25)
: indicates that the conversation is not in **Initialize** state. Therefore, the *conversation_type* characteristic cannot be altered.

Results of the Call

If a *return_code* other than CM_OK is returned on the call, the *conversation_type* characteristic is not changed. This call does not cause a state change.

Adding CMSCT to Our Requester Program

We have avoided basic conversations up to now because they are more difficult to write. However, just for this section, we will start one to demonstrate how the changes we have made to the server program work.

We are not going to worry about setting up a program to correctly process a basic conversation. We will just add a Set_Conversation_Type call in the PROCESS EXEC, following the Initialize_Conversation (CMINIT) call.

We will want to continue focusing on mapped conversations, so we will remove the Set_Conversation_Type call after we have tested the exec.

The requester program will temporarily include these additions:

```
/*====================================================================*/
/*  PROCESS EXEC - Sample file requester application.                 */
/*====================================================================*/

  ⋮
/*--------------------------------------------------------------------*/
/* Initialize the conversation.                                       */
/*--------------------------------------------------------------------*/
'CMINIT conversation_ID sym_dest_name return_code'
say; say 'Routine called:  CMINIT'
```

```
if (return_code ¬= CM_OK) then call ErrorHandler 'CMINIT'
call TraceParms
/*-------------------------------------------------------------------*/
/* Set the conversation_type to basic.                               */
/*-------------------------------------------------------------------*/
conversation_type = CM_BASIC_CONVERSATION
'CMSCT conversation_ID conversation_type return_code'
say; say 'Routine called:  CMSCT'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMSCT'
call TraceParms 'conversation_ID conversation_type return_code'
/*-------------------------------------------------------------------*/
/* Determine if confirmation processing is desired.                  */
/*-------------------------------------------------------------------*/
say; say 'Would you like confirmation processing? (Y/N)'
parse upper pull perform_confirm
if (perform_confirm = 'Y') then
  do
    /*-----------------------------------------------------------------*/
    /* Set sync_level to CM_CONFIRM.                                   */
    /*-----------------------------------------------------------------*/
    sync_level = CM_CONFIRM
    'CMSSL conversation_ID sync_level return_code'
    say; say 'Routine called:  CMSSL'
    if (return_code ¬= CM_OK) then call ErrorHandler 'CMSSL
    call TraceParms
    say '  Confirmation processing enabled'
  end
   .
   .
   .

/*------------------------- Subroutines --------------------------*/

TraceParms:
/*-------------------------------------------------------------------*/
/* Display parameters and their values as passed to this subroutine.*/
/*-------------------------------------------------------------------*/
parse arg parmlist
do word_num = 1 to words(parmlist)
  parameter = word(parmlist,word_num)
  select
    when (parameter = 'return_code') then
      say '  return_code is' cm_return_code.return_code
    when (parameter = 'buffer') then
      say '  buffer is' left(buffer,send_length)
    when (parameter = 'receive_buffer') then
      say '  buffer is' left(receive_buffer,received_length)
    when (parameter = 'data_received') then
      say '  data_received is' cm_data_received.data_received
    when (parameter = 'status_received') then
      say '  status_received is' cm_status_received.status_received
    when (parameter = 'request_to_send_received') then
      say '  request_to_send_received is',
          cm_request_to_send_received.request_to_send_received
    when (parameter = 'sync_level') then
      say '  sync_level is' cm_sync_level.sync_level
    when (parameter = 'prepare_to_receive_type') then
      say '  prepare_to_receive_type is',
          cm_prepare_to_receive_type.prepare_to_receive_type
    when (parameter = 'deallocate_type') then
      say '  deallocate_type is' cm_deallocate_type.deallocate_type
    when (parameter = 'conversation_type') then
      say '  conversation_type is',
          cm_conversation_type.conversation_type
    otherwise
      say ' ' parameter 'is' value(parameter)
  end
end
/*-------------------------------------------------------------------*/
```

```
/* Extract the current conversation state of the local program.     */
/*------------------------------------------------------------------*/
/* Commenting out next four lines ...
'CMECS conversation_ID conversation_state return_code'
if (return_code = CM_OK) then
  say '  conversation_state is =>',
      cm_conversation_state.conversation_state
... */

return
   :
```

File the exec and start it up with

```
process getfile
```

and choose not to have confirmation performed.

Results on the requester side should be:

```
process getfile
Requesting the file:  TEST FILE A

Routine called:  CMINIT

Routine called:  CMSCT
  conversation_ID is 00000000
  conversation_type is CM_BASIC_CONVERSATION
  return_code is CM_OK

Would you like confirmation processing? (Y/N)
N

Routine called:  CMALLC

Routine called:  CMSEND

* ERROR:  An error occurred during a CMSEND call
          The return_code was set to CM_PROGRAM_ERROR_PURGING

Routine called:  CMSDT

Routine called:  CMDEAL
Ready;
```

Figure 44. Results of PROCESS EXEC Establishing a Basic Conversation

The server's results should be:

```
Routine called:  CMACCP

Routine called:  CMECT
  conversation_ID is 00000000
  conversation_type is CM_BASIC_CONVERSATION
  return_code is CM_OK

* ERROR:  Accepting and deallocating a basic conversation

Routine called:  CMSERR
  conversation_ID is 00000000
  request_to_send_received is CM_REQ_TO_SEND_NOT_RECEIVED
  return_code is CM_OK

Routine called:  CMSDT

Routine called:  CMDEAL
Ready;
```

Figure 45. Results of SENDBACK EXEC Detecting a Basic Conversation

The requester program sets the conversation type to basic and the server program detects this with the Extract_Conversation_Type call. The server program issues a Send_Error (CMSERR) call and then a Deallocate (CMDEAL) with *deallocate_type* set to CM_DEALLOCATE_ABEND. As the Send_Error flows to the requester, it causes the incoming requested file name data to be purged without being received. That fact is reported to the requester when its Send_Data (CMSEND) call completes with a *return_code* of CM_PROGRAM_ERROR_PURGING.

Because Send_Error causes the side of the conversation that calls it to enter **Send** state, its partner (the requester program in this case) enters **Receive** state when its Send_Data call completes. The requester recognizes that the last *return_code* value is not CM_OK, so it calls ErrorHandler to terminate the conversation. Although both partners issue a Deallocate call, only one of them will be successful. We chose not to have our ErrorHandler subroutine report any additional errors because it is already deallocating the conversation and terminating the program.

Now you can go back and remove the code we added for the Set_Conversation_Type call, including that in the TraceParms subroutine, from PROCESS EXEC.

The Set_Partner_LU_Name (CMSPLN) Call

The Set_Partner_LU_Name (CMSPLN) call sets the *partner_LU_name* characteristic for a given conversation, overriding the partner LU name obtained from side information using the *sym_dest_name*.

This call does not change any data in the side information, and the new *partner_LU_name* value will be known only for this particular conversation.

Only the program initializing a conversation (using the Initialize_Conversation (CMINIT) call) can issue Set_Partner_LU_Name. The call must be issued while in **Initialize** state, prior to the Allocate (CMALLC) call for the specified conversation.

Partner location information is usually kept in side information. This call might be included if a particular program did not want to use the *partner_LU_name* acquired from side information, or if the program wanted to ensure that the *partner_LU_name* it used would not be affected by a change to the `:luname.` tag in the

communications directory. Explicitly setting the *partner_LU_name* may decrease the portability of the program to other SAA platforms because VM uses a space as a delimiter rather than a period.

The format for Set_Partner_LU_Name is:

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

Input Parameters

Use the ***conversation_ID*** parameter to identify the conversation.

Use the ***partner_LU_name*** parameter to specify the name of the remote LU where the remote transaction program is located. This LU name is any name by which the local LU knows the remote LU for purposes of allocating a conversation.

In VM, a *partner_LU_name* consists of two character strings separated by a blank.

Use the ***partner_LU_name_length*** parameter to specify the length of the partner LU name, which can be from 1 to 17 bytes.

Output Parameter

Possible values for the ***return_code*** parameter are:

CM_OK (0)
: indicates that the partner LU name has been set.

CM_PRODUCT_SPECIFIC_ERROR (20)
: indicates a CMS error; check the CPICOMM LOGDATA file.

CM_PROGRAM_PARAMETER_CHECK (24)
: indicates that the specified conversation ID is unassigned or that the specified *partner_LU_name_length* is less than 1 or greater than 17.

CM_PROGRAM_STATE_CHECK (25)
: indicates that the conversation is not in **Initialize** state. Therefore, the *partner_LU_name* conversation characteristic cannot be altered.

Results of the Call

If the return_code is not CM_OK, the *partner_LU_name* and *partner_LU_name_length* characteristics remain unchanged. This call does not cause a state change.

Adding CMSPLN to Our Requester Program

Our requester program does not need the Set_Partner_LU_Name call, but let's try it out to see how it works. We will add the call following the Initialize_Conversation call and provide an invalid value.

Your exec should now have the following lines:

```
/*====================================================================*/
/*  PROCESS EXEC - Sample file requester application.                 */
/*====================================================================*/

  ⋮
/*--------------------------------------------------------------------*/
/* Initialize the conversation.                                       */
/*--------------------------------------------------------------------*/
'CMINIT conversation_ID sym_dest_name return_code'
```

```
say; say 'Routine called:  CMINIT'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMINIT'
call TraceParms
/*-------------------------------------------------------------------*/
/* Set the partner_LU_name explicitly.                               */
/*-------------------------------------------------------------------*/
partner_LU_name = 'UNKNOWN NAME'
partner_LU_name_length = length(partner_LU_name)
'CMSPLN conversation_ID partner_LU_name',
       'partner_LU_name_length return_code'
say; say 'Routine called:  CMSPLN'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMSPLN'
call TraceParms 'conversation_ID partner_LU_name',
                'partner_LU_name_length return_code'
/*-------------------------------------------------------------------*/
/* Determine if confirmation processing is desired.                  */
/*-------------------------------------------------------------------*/
say; say 'Would you like confirmation processing? (Y/N)'
parse upper pull perform_confirm
if (perform_confirm = 'Y') then
  do
    /*---------------------------------------------------------------*/
    /* Set sync_level to CM_CONFIRM.                                 */
    /*---------------------------------------------------------------*/
    sync_level = CM_CONFIRM
    'CMSSL conversation_ID sync_level return_code'
    say; say 'Routine called:  CMSSL'
    if (return_code ¬= CM_OK) then call ErrorHandler 'CMSSL'
    call TraceParms 'conversation_ID sync_level return_code'
    say '  Confirmation processing enabled'
  end
  .
  .
```

File the exec and execute it by entering

```
process getfile
```

from the command line and answer the confirmation prompt with 'N'.

The results on the requester side should be:

```
process getfile
Requesting the file:  TEST FILE A

Routine called:  CMINIT

Routine called:  CMSPLN
  conversation_ID is 00000000
  partner_LU_name is UNKNOWN NAME
  partner_LU_name_length is 12
  return_code is CM_OK

Would you like confirmation processing? (Y/N)
N

Routine called:  CMALLC

* ERROR:  An error occurred during a CMALLC call
          The return_code was set to CM_PARAMETER_ERROR

Routine called:  CMSDT

Routine called:  CMDEAL
Ready;
```

Figure 46. Results of Setting an Unknown LU Name from PROCESS EXEC

The Set_Partner_LU_Name call executed correctly, but the Allocate call failed with *return_code* set to CM_PARAMETER_ERROR. The problem is that we specified a bad value for the *partner_LU_name*.

The failure of our program has shown that inclusion of the Set_Partner_LU_Name call resulted in overriding the side information value for the partner LU name, as we expected.

You can either correct the *partner_LU_name* or remove the call to Set_Partner_LU_Name. We will go back and provide the valid *partner_LU_name* for our server. (Remember that you need to substitute the appropriate name if you are using a different user ID.) For the example we have shown, we need to replace UNKNOWN NAME with *USERID SERVR, as follows:

```
partner_LU_name = '*USERID SERVR'
```

See the *z/VM: Connectivity* book for information on LU naming conventions in VM.

The exec should work correctly again, if you want to execute it.

The Set_TP_Name (CMSTPN) Call

The Set_TP_Name (CMSTPN) call sets the *TP_name* characteristic for a given conversation, overriding the transaction program (TP) name obtained from side information using the *sym_dest_name*.

Executing this call does not change the transaction program name provided with the `:tpn.` tag in the communications directory. It only changes the value of the *TP_name* characteristic for this particular conversation.

Only the program initializing a conversation (using the Initialize_Conversation (CMINIT) call) can issue Set_TP_Name. The call must be issued while in **Initialize** state, prior to the Allocate (CMALLC) call for the specified conversation.

Note: The TP name must be formatted according to the naming conventions of the partner LU.

The format for Set_TP_Name is:

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

Input Parameters

Use the ***conversation_ID*** parameter to identify the conversation.

Use the ***TP_name*** parameter to specify the name of the remote program, as it is known at the target LU.

Use the ***TP_name_length*** parameter to specify the length of the *TP_name*, from 1 to 64 bytes.

Output Parameter

Possible values for the ***return_code*** parameter are:

CM_OK (0)
: indicates that the TP name has been set.

CM_PRODUCT_SPECIFIC_ERROR (20)
: indicates a CMS error; check the CPICOMM LOGDATA file.

CM_PROGRAM_PARAMETER_CHECK (24)
: indicates that the conversation ID is unassigned or the specified length of the TP name is less than 1 or greater than 64.

CM_PROGRAM_STATE_CHECK (25)
: indicates that the conversation is not in **Initialize** state. Therefore, the *TP_name* characteristic cannot be altered.

Results of the Call

If the return code is not CM_OK, then the *TP_name* and *TP_name_length* characteristics remain unchanged. This call does not cause a state change.

Adding CMSTPN to Our Requester Program

Again, our requester program does not need to use Set_TP_Name, but we will add the call to demonstrate its use. Let's add it following the Set_Partner_LU_Name (CMSPLN) call and provide an invalid value.

Your exec should now have the following lines.

```
/*====================================================================*/
/*  PROCESS EXEC - Sample file requester application.                 */
/*====================================================================*/

   ⋮
/*--------------------------------------------------------------------*/
/* Set the partner_LU_name explicitly.                                */
/*--------------------------------------------------------------------*/
partner_LU_name = '*USERID SERVR'
partner_LU_name_length = length(partner_LU_name)
'CMSPLN conversation_ID partner_LU_name',
       'partner_LU_name_length return_code'
say; say 'Routine called:  CMSPLN'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMSPLN'
call TraceParms
/*--------------------------------------------------------------------*/
```

```
/* Set the transaction program name (TP_name) explicitly.          */
/*-----------------------------------------------------------------*/
TP_name = 'NOTATPNAME'
TP_name_length = length(TP_name)
'CMSTPN conversation_ID TP_name TP_name_length return_code'
say; say 'Routine called:  CMSTPN'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMSTPN'
call TraceParms 'conversation_ID TP_name TP_name_length return_code'
/*-----------------------------------------------------------------*/
/* Determine if confirmation processing is desired.                */
/*-----------------------------------------------------------------*/
say; say 'Would you like confirmation processing? (Y/N)'
parse upper pull perform_confirm
if (perform_confirm = 'Y') then
  do
    /*-------------------------------------------------------------*/
    /* Set sync_level to CM_CONFIRM.                               */
    /*-------------------------------------------------------------*/
    sync_level = CM_CONFIRM
    'CMSSL conversation_ID sync_level return_code'
    say; say 'Routine called:  CMSSL'
    if (return_code ¬= CM_OK) then call ErrorHandler 'CMSSL'
    call TraceParms
    say '  Confirmation processing enabled'
  end
   ⋮
```

Now to test what happens, begin the program with

```
process getfile
```

and again forgo the confirmation processing.

The results on the requester's side are:

```
process getfile
Requesting the file:  TEST FILE A

Routine called:  CMINIT

Routine called:  CMSPLN

Routine called:  CMSTPN
  conversation_ID is 00000000
  TP_name is NOTATPNAME
  TP_name_length is 10
  return_code is CM_OK

Would you like confirmation processing? (Y/N)
N

Routine called:  CMALLC

Routine called:  CMSEND

* ERROR:  An error occurred during a CMSEND call
          The return_code was set to CM_TPN_NOT_RECOGNIZED

Routine called:  CMSDT
Ready;
```

Figure 47. Results of Setting an Incorrect TP Name from PROCESS EXEC

At the server's terminal, you will see:

```
hh:mm:ss  * MSG FROM SERVR  : DMSIUH2027E Connection request on path 0
is severed for reason = 7
```

Figure 48. Results on Server Virtual Machine Because of an Incorrect TP Name

The results should look familiar to you because we had a similar problem in Chapter 2, "Starter Set CPI Communications Calls." The partner LU was unable to start a program with the TP name we provided because it does not exist. The sever reported at the server's terminal is reflected back to the requester on the Send_Data (CMSEND) call when it completes with *return_code* set to CM_TPN_NOT_RECOGNIZED. The conversation has entered **Reset** state at the requester's end when the Send_Data call completes, so the Set_Deallocate_Type (CMSDT) call fails with CM_PROGRAM_PARAMETER_CHECK, which we do not see, and the Deallocate (CMDEAL) is not issued.

This problem can be remedied easily by either replacing NOTATPNAME with the correct value or removing the call. We will replace NOTATPNAME with the correct value as follows :

```
TP_name = 'GET'
```

Feel free to run the programs again to confirm that they work correctly after making the change.

Overviews of Additional Advanced Calls

Although we will not be adding the routines covered in this section to either of our programs, they are included here to give you a brief introduction to some additional CPI Communications calls that are available to programmers.

Extract_Mode_Name (CMEMN) Call

The Extract_Mode_Name (CMEMN) call extracts the mode name for a conversation. CPI Communications returns the mode name in the *mode_name* parameter.

Extract_Partner_LU_Name (CMEPLN) Call

The Extract_Partner_LU_Name (CMEPLN) call extracts the partner LU name for a conversation. CPI Communications returns the partner LU name in the *partner_LU_name* parameter.

Extract_Mode_Name (CMEMN) and Extract_Partner_LU_Name can provide information about the session carrying the conversation and the conversation originator.

Extract_Sync_Level (CMESL) Call

The Extract_Sync_Level (CMESL) call extracts the value of the *sync_level* conversation characteristic for a given conversation. The value is returned in the *sync_level* parameter.

Request_To_Send (CMRTS) Call

A program can use the Request_To_Send (CMRTS) call to notify its conversation partner that it wants to enter **Send** state for a given conversation.

When a conversation is in **Receive** state, it cannot send data without permission. The partner in **Send** state effectively exercises control over the conversation. The Request_To_Send call is used by a program in **Receive** state to notify its partner that it wishes to change states and send data.

The partner program is made aware of the request by the *request_to_send_received* parameter being set to CM_REQ_TO_SEND_RECEIVED on a Send_Data (CMSEND), Test_Request_To_Send (CMTRTS), Confirm (CMCFM), Send_Error (CMSERR), or Receive (CMRCV) call. The Request_To_Send call is the only way for a program to request control.

The program that issues Request_To_Send does not get control of the conversation until it receives a *status_received* value of CM_SEND_RECEIVED or CM_CONFIRM_SEND_RECEIVED from the remote program on a subsequent Receive call.

Set_Error_Direction (CMSED) Call

The Set_Error_Direction (CMSED) call sets the *error_direction* characteristic for a given conversation, overriding the value assigned by the Initialize_Conversation (CMINIT) or Accept_Conversation (CMACCP) call.

A program should issue Set_Error_Direction before calling Send_Error (CMSERR) for a conversation in **Send-Pending** state. **Send-Pending** state arises when a Receive (CMRCV) call completes with both data and a conversation status of CM_SEND_RECEIVED. This call lets a program indicate to its partner whether the error is in the data just received, or is a local processing error.

Set_Fill (CMSF) Call

The Set_Fill (CMSF) call sets the *fill* characteristic for a given conversation, overriding the value assigned with the Initialize_Conversation (CMINIT) or Accept_Conversation (CMACCP) calls. The Set_Fill call is valid only for basic conversations.

Use the Set_Fill call to specify that you want to receive data independent of its logical record format. In other words, each logical record will not necessarily be presented to your program as it arrives, but rather will be buffered. The amount of data received will be equal to or less than the length specified by the *requested_length* parameter of the Receive call.

Set_Log_Data (CMSLD) Call

The Set_Log_Data (CMSLD) call sets the *log_data* and *log_data_length* characteristics for a given conversation, overriding the values assigned with the Initialize_Conversation (CMINIT) or Accept_Conversation (CMACCP) calls. The Set_Log_Data call is valid only for basic conversations.

Log data is program-unique error information that is to be logged. The data supplied by the program is any data the program wants to have logged, such as information that can help identify the cause of the error. The data is sent on a Send_Error call or a Deallocate call when *deallocate_type* is CM_DEALLOCATE_ABEND.

Set_Mode_Name (CMSMN) Call

The Set_Mode_Name (CMSMN) call sets the *mode_name* and *mode_name_length* characteristics for a given conversation, overriding the value originally obtained from

side information using the *sym_dest_name*. The mode name designates network properties for the session to be allocated for the conversation. Network properties include, for example, the class of service to be used and whether data is to be encrypted. The mode name is needed only when allocating a conversation to a partner in the SNA network.

Only the program initiating a conversation (using the Initialize_Conversation (CMINIT) call) can issue Set_Mode_Name. The call must be issued while in **Initialize** state, prior to the Allocate (CMALLC) call for the specified conversation.

As with the Set_Partner_LU_Name (CMSPLN) and Set_TP_Name (CMSTPN) calls, a program would use Set_Mode_Name to avoid dependency on the side information. As was the case with the other routines, explicitly setting the *mode_name* within a program may make that program less portable.

Set_Return_Control (CMSRC) Call

The Set_Return_Control (CMSRC) call sets the *return_control* characteristic for a given conversation, overriding the value assigned with the Initialize_Conversation (CMINIT) call.

Set_Return_Control can be called only by the program that initiates a conversation (using the Initialize_Conversation call). The call must be issued while in **Initialize** state, prior to the Allocate (CMALLC) call for the specified conversation.

A program might use this call to set the *return_control* characteristic to CM_IMMEDIATE if it had other processing it could perform should a wait be required for a session to become available. If the Allocate call completes with *return_code* set to CM_UNSUCCESSFUL, the program could do some other processing until it wanted to attempt another conversation allocation. Not setting the *return_control* characteristic to CM_IMMEDIATE would result in the program waiting until a session was available for the conversation to be allocated.

Set_Receive_Type (CMSRT) Call

The Set_Receive_Type (CMSRT) call sets the *receive_type* conversation characteristic for a given conversation and overrides the initial value assigned by the Initialize_Conversation (CMINIT) or Accept_Conversation (CMACCP) call.

With the default *receive_type* setting of CM_RECEIVE_AND_WAIT, a Receive (CMRCV) call will not complete until incoming information is available for it to receive.

If a program needs to perform processing outside of a CPI Communications conversation, changing the *receive_type* to CM_IMMEDIATE could be beneficial. That way, the program could periodically issue a Receive call to check for incoming information, and if there was not anything available to receive, the call would complete with the *data_received* parameter set to CM_NO_DATA_RECEIVED and the *status_received* parameter set to CM_NO_STATUS_RECEIVED. The program could then perform some of its other processing until it again wanted to poll for information.

The Set_Receive_Type call only affects the local side of the conversation, and after it is set, the *receive_type* will remain in effect for the rest of the conversation or until Set_Receive_Type is called again.

Test_Request_To_Send_Received (CMTRTS) Call

A program uses the Test_Request_To_Send_Received (CMTRTS) call to determine whether a request-to-send notification has been received from the remote program for a given conversation.

The Modified Sample Execs

In this section, we have included a complete listing of both of the programs we have been developing, incorporating all the changes made through the end of this chapter.

The PROCESS Sample File Requester Exec

```
/*====================================================================*/
/*  PROCESS EXEC - Sample file requester application.                 */
/*====================================================================*/

arg sym_dest_name fname ftype fmode .           /* get user's input */
/*--------------------------------------------------------------------*/
/* If a file was not specifically requested, set up a default.        */
/*--------------------------------------------------------------------*/
if (fname = '') then
  do
    fname = 'TEST'
    ftype = 'FILE'
    fmode = 'A'
  end
say 'Requesting the file: ' fname ftype fmode
/*--------------------------------------------------------------------*/
/* Set up REXX environment for program-to-program communications      */
/* and enable error trapping of REXX errors.                          */
/*--------------------------------------------------------------------*/
address cpicomm
signal on error
/*--------------------------------------------------------------------*/
/* Equate pseudonyms to their integer values based on the             */
/* definitions contained in the CMREXX COPY file.                     */
/*--------------------------------------------------------------------*/
address command 'EXECIO * DISKR CMREXX COPY * (FINIS STEM PSEUDONYM.'
do index = 1 to pseudonym.0
  interpret pseudonym.index
end
/*--------------------------------------------------------------------*/
/* Initialize the conversation.                                       */
/*--------------------------------------------------------------------*/
'CMINIT conversation_ID sym_dest_name return_code'
say; say 'Routine called:  CMINIT'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMINIT'
call TraceParms
/*--------------------------------------------------------------------*/
/* Set the partner_LU_name explicitly.                                */
/*--------------------------------------------------------------------*/
partner_LU_name = '*USERID SERVR'
partner_LU_name_length = length(partner_LU_name)
'CMSPLN conversation_ID partner_LU_name',
       'partner_LU_name_length return_code'
say; say 'Routine called:  CMSPLN'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMSPLN'
call TraceParms
/*--------------------------------------------------------------------*/
/* Set the transaction program name (TP_name) explicitly.             */
/*--------------------------------------------------------------------*/
TP_name = 'GET'
TP_name_length = length(TP_name)
```

```
'CMSTPN conversation_ID TP_name TP_name_length return_code'
say; say 'Routine called:  CMSTPN'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMSTPN'
call TraceParms
/*-------------------------------------------------------------------*/
/* Determine if confirmation processing is desired.                  */
/*-------------------------------------------------------------------*/
say; say 'Would you like confirmation processing? (Y/N)'
parse upper pull perform_confirm
if (perform_confirm = 'Y') then
  do
    /*---------------------------------------------------------------*/
    /* Set sync_level to CM_CONFIRM.                                 */
    /*---------------------------------------------------------------*/
    sync_level = CM_CONFIRM
    'CMSSL conversation_ID sync_level return_code'
    say; say 'Routine called:  CMSSL'
    if (return_code ¬= CM_OK) then call ErrorHandler 'CMSSL'
    call TraceParms
    say '  Confirmation processing enabled'
  end
/*-------------------------------------------------------------------*/
/* Allocate the conversation.                                        */
/*-------------------------------------------------------------------*/
'CMALLC conversation_ID return_code'
say; say 'Routine called:  CMALLC'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMALLC'
call TraceParms
/*-------------------------------------------------------------------*/
/* Send the name of the file being requested to the partner program.*/
/*-------------------------------------------------------------------*/
buffer = fname ftype fmode
send_length = length(buffer)
'CMSEND conversation_ID buffer send_length',
       'request_to_send_received return_code'
say; say 'Routine called:  CMSEND'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMSEND'
call TraceParms
/*-------------------------------------------------------------------*/
/* Call Confirm only when sync_level is not CM_NONE.  We can use     */
/* the confirmation processing flag set from console input.          */
/*-------------------------------------------------------------------*/
if (perform_confirm = 'Y') then
  do
    /*---------------------------------------------------------------*/
    /* Confirm that partner has started and received the name of     */
    /* the requested file.                                           */
    /*---------------------------------------------------------------*/
    'CMCFM conversation_ID request_to_send_received',
          'return_code'
    say; say 'Routine called:  CMCFM'
    if (return_code ¬= CM_OK) then call ErrorHandler 'CMCFM
    call TraceParms
  end
/*-------------------------------------------------------------------*/
/* Set the prepare_to_receive_type to CM_PREP_TO_RECEIVE_FLUSH.      */
/*-------------------------------------------------------------------*/
prepare_to_receive_type = CM_PREP_TO_RECEIVE_FLUSH
'CMSPTR conversation_ID prepare_to_receive_type return_code'
say; say 'Routine called:  CMSPTR'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMSPTR'
call TraceParms
/*-------------------------------------------------------------------*/
/* Issue Prepare_To_Receive to switch the conversation state from    */
/* Send state to Receive state.                                      */
/*-------------------------------------------------------------------*/
'CMPTR conversation_ID return_code'
```

```
say; say 'Routine called:  CMPTR'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMPTR'
call TraceParms
/*-----------------------------------------------------------------*/
/* Start a Receive loop.  Receive calls will be issued until       */
/* notification that the partner has finished sending data and     */
/* entered Receive state at its end of the conversation (noted by  */
/* receipt of CM_SEND_RECEIVED or CM_CONFIRM_SEND_RECEIVED         */
/* for status_received) or until a return_code value other than    */
/* CM_OK is returned.  The record length of the incoming data      */
/* is assumed to be 80 bytes, or less.                             */
/*-----------------------------------------------------------------*/
complete_line = ''
requested_length = 80
do until (status_received = CM_SEND_RECEIVED) |,
         (status_received = CM_CONFIRM_SEND_RECEIVED)
  /*---------------------------------------------------------------*/
  /* Receive information from the conversation partner.            */
  /*---------------------------------------------------------------*/
  'CMRCV conversation_ID receive_buffer requested_length',
        'data_received received_length status_received',
        'request_to_send_received return_code'
  say; say 'Routine called:  CMRCV'
  select
    when (return_code = CM_OK) then
      do
        call TraceParms 'data_received status_received'
        if (data_received ¬= CM_NO_DATA_RECEIVED) then
          do
            receive_buffer = left(receive_buffer,received_length)
            complete_line = complete_line || receive_buffer
          end
        if (data_received = CM_COMPLETE_DATA_RECEIVED) then
          do
            /*-------------------------------------------------------*/
            /* Use EXECIO to write the data to OUTPUT LOGFILE A      */
            /* and reset the complete_line variable to nulls.        */
            /*-------------------------------------------------------*/
            address command 'EXECIO 1 DISKW OUTPUT LOGFILE A (FINIS',
                            'STRING' complete_line
            complete_line = ''
          end
        /*-----------------------------------------------------------*/
        /* Determine whether a confirmation request has been         */
        /* received.  If so, respond with a positive reply.          */
        /*-----------------------------------------------------------*/
        if (status_received = CM_CONFIRM_RECEIVED) |,
           (status_received = CM_CONFIRM_SEND_RECEIVED) |,
           (status_received = CM_CONFIRM_DEALLOC_RECEIVED) then
          do
            /*-------------------------------------------------------*/
            /* Issue Confirmed to reply to the partner.              */
            /*-------------------------------------------------------*/
            'CMCFMD conversation_ID return_code'
            say; say 'Routine called:  CMCFMD'
            if (return_code ¬= CM_OK) then call ErrorHandler 'CMCFMD'
            call TraceParms
          end
      end
    otherwise
      call ErrorHandler 'CMRCV'
  end
end
/*-----------------------------------------------------------------*/
/* Deallocate the conversation normally.                           */
/*-----------------------------------------------------------------*/
'CMDEAL conversation_ID return_code'
```

```
say; say 'Routine called:  CMDEAL'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMDEAL'
call TraceParms

GetOut:
  exit

/*-------------------------- Subroutines --------------------------*/

TraceParms:
/*-----------------------------------------------------------------*/
/* Display parameters and their values as passed to this subroutine.*/
/*-----------------------------------------------------------------*/
parse arg parmlist
do word_num = 1 to words(parmlist)
  parameter = word(parmlist,word_num)
  select
    when (parameter = 'return_code') then
      say '  return_code is' cm_return_code.return_code
    when (parameter = 'buffer') then
      say '  buffer is' left(buffer,send_length)
    when (parameter = 'receive_buffer') then
      say '  buffer is' left(receive_buffer,received_length)
    when (parameter = 'data_received') then
      say '  data_received is' cm_data_received.data_received
    when (parameter = 'status_received') then
      say '  status_received is' cm_status_received.status_received
    when (parameter = 'request_to_send_received') then
      say '  request_to_send_received is',
          cm_request_to_send_received.request_to_send_received
    when (parameter = 'sync_level') then
      say '  sync_level is' cm_sync_level.sync_level
    when (parameter = 'prepare_to_receive_type') then
      say '  prepare_to_receive_type is',
          cm_prepare_to_receive_type.prepare_to_receive_type
    when (parameter = 'deallocate_type') then
      say '  deallocate_type is' cm_deallocate_type.deallocate_type
    otherwise
      say ' ' parameter 'is' value(parameter)
  end
end
/*-----------------------------------------------------------------*/
/* Extract the current conversation state of the local program.     */
/*-----------------------------------------------------------------*/
/* Commenting out next four lines ...
'CMECS conversation_ID conversation_state return_code'
if (return_code = CM_OK) then
  say '  conversation_state is =>',
      cm_conversation_state.conversation_state
... */

return


Error:
/*-----------------------------------------------------------------*/
/* Report error when REXX special variable RC is not 0.            */
/*-----------------------------------------------------------------*/
say
say '* ERROR:  REXX has detected an error'
say '          The return code variable RC was set to' rc
call AbnormalEnd
signal GetOut


ErrorHandler:
/*-----------------------------------------------------------------*/
```

```
/* Report routine that failed and the error return code.           */
/*-----------------------------------------------------------------*/
parse arg routine_name
say
say '* ERROR:  An error occurred during a' routine_name 'call'
say '          The return_code was set to' cm_return_code.return_code
call AbnormalEnd
signal GetOut


AbnormalEnd:
/*-----------------------------------------------------------------*/
/* Abnormally deallocate the conversation.  Since we are exiting   */
/* due to an error, we will not display an error message if the    */
/* Set_Deallocate_Type or Deallocate call encounters an error.     */
/*-----------------------------------------------------------------*/
deallocate_type = CM_DEALLOCATE_ABEND
'CMSDT conversation_ID deallocate_type return_code'
say; say 'Routine called:  CMSDT'
if (return_code = CM_OK) then
  do
    call TraceParms
    'CMDEAL conversation_ID return_code'
    say; say 'Routine called:  CMDEAL'
    if (return_code = CM_OK) then
      call TraceParms
  end

return
```

The SENDBACK Sample Server Exec

```
/*=================================================================*/
/*  SENDBACK EXEC - Sample server application.                     */
/*=================================================================*/

/*-----------------------------------------------------------------*/
/* Set up REXX environment for program-to-program communications   */
/* and enable error trapping of REXX errors.                       */
/*-----------------------------------------------------------------*/
address cpicomm
signal on error
/*-----------------------------------------------------------------*/
/* Equate pseudonyms to their integer values based on the          */
/* definitions contained in the CMREXX COPY file.                  */
/*-----------------------------------------------------------------*/
address command 'EXECIO * DISKR CMREXX COPY * (FINIS STEM PSEUDONYM.'
do index = 1 to pseudonym.0
  interpret pseudonym.index
end
/*-----------------------------------------------------------------*/
/* Accept the incoming conversation.                               */
/*-----------------------------------------------------------------*/
'CMACCP conversation_ID return_code'
say; say 'Routine called:  CMACCP'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMACCP'
call TraceParms
/*-----------------------------------------------------------------*/
/* Extract conversation_type to ensure the conversation is mapped.  */
/*-----------------------------------------------------------------*/
'CMECT conversation_ID conversation_type return_code'
say; say 'Routine called:  CMECT'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMECT'
call TraceParms
/*-----------------------------------------------------------------*/
/* If the conversation is basic, deallocate abnormally.            */
/*-----------------------------------------------------------------*/
```

```
if (conversation_type = CM_BASIC_CONVERSATION) then
  do
    say; say '* ERROR:  Accepting and deallocating a basic',
             'conversation'
    /*-----------------------------------------------------------------*/
    /* Call Send_Error to notify partner that error was detected.   */
    /* Since the program is going to exit, do not check the         */
    /* Send_Error results for an error.                             */
    /*-----------------------------------------------------------------*/
    'CMSERR conversation_ID request_to_send_received return_code'
    say; say 'Routine called:  CMSERR'
    if (return_code = CM_OK) then
      call TraceParms
    call AbnormalEnd
    signal GetOut
  end
/*-------------------------------------------------------------------*/
/* Start a Receive loop.                                             */
/* Receive data, status, or both from conversation partner.          */
/*-------------------------------------------------------------------*/
requested_file = ''
requested_length = 20
do until (CMRCV_return_code ¬= CM_OK) |,
         (status_received = CM_CONFIRM_DEALLOC_RECEIVED)
  'CMRCV conversation_ID receive_buffer requested_length',
        'data_received received_length status_received',
        'request_to_send_received return_code'
  CMRCV_return_code = return_code
  say; say 'Routine called:  CMRCV'
  select
    when (CMRCV_return_code = CM_OK) then
      do
        call TraceParms 'data_received status_received'
        if (data_received ¬= CM_NO_DATA_RECEIVED) then
          do
            receive_buffer = left(receive_buffer,received_length)
            requested_file = requested_file || receive_buffer
          end
        /*-------------------------------------------------------------*/
        /* Determine whether a confirmation request has been        */
        /* received.  If so, respond with a positive reply.         */
        /*-------------------------------------------------------------*/
        if (status_received = CM_CONFIRM_RECEIVED) |,
           (status_received = CM_CONFIRM_SEND_RECEIVED) |,
           (status_received = CM_CONFIRM_DEALLOC_RECEIVED) then
          do
            /*---------------------------------------------------------*/
            /* Issue Confirmed to reply to the partner.             */
            /*---------------------------------------------------------*/
            'CMCFMD conversation_ID return_code'
            say; say 'Routine called:  CMCFMD'
            if (return_code ¬= CM_OK) then call ErrorHandler 'CMCFMD'
            call TraceParms
          end
        if (status_received = CM_SEND_RECEIVED) |,
           (status_received = CM_CONFIRM_SEND_RECEIVED) then
          call SendFile
        else
          if (status_received = CM_CONFIRM_DEALLOC_RECEIVED) then
            do
              say; say 'Conversation deallocated by partner'
            end
      end
    when (CMRCV_return_code = CM_DEALLOCATED_NORMAL) then
      do
        call TraceParms 'data_received status_received'
        say; say 'Conversation deallocated by partner'
```

```
      end
    otherwise
      call ErrorHandler 'CMRCV'
  end
end

GetOut:
  exit

/*-------------------------- Subroutines --------------------------*/

SendFile:
/*---------------------------------------------------------------*/
/* Read the contents of the requested file and send each line of */
/* the file to the partner program.                              */
/*---------------------------------------------------------------*/
address command 'EXECIO * DISKR' requested_file '(FINIS STEM LINE.'
do index = 1 to line.0
  if (index = line.0) then
    /*-----------------------------------------------------------*/
    /* Reset the send_type conversation characteristic just      */
    /* before the final Send_Data call.                          */
    /*-----------------------------------------------------------*/
    do
      send_type = CM_SEND_AND_PREP_TO_RECEIVE
      'CMSST conversation_ID send_type return_code'
      say; say 'Routine called:  CMSST'
      if (return_code ¬= CM_OK) then call ErrorHandler 'CMSST'
      call TraceParms
    end
  buffer = line.index
  send_length = length(buffer)
  'CMSEND conversation_ID buffer send_length',
         'request_to_send_received return_code'
  say; say 'Routine called:  CMSEND'
  if (return_code ¬= CM_OK) then call ErrorHandler 'CMSEND'
  call TraceParms
end

return


TraceParms:
/*---------------------------------------------------------------*/
/* Display parameters and their values as passed to this subroutine.*/
/*---------------------------------------------------------------*/
parse arg parmlist
do word_num = 1 to words(parmlist)
  parameter = word(parmlist,word_num)
  select
    when (parameter = 'return_code') then
      say '  return_code is' cm_return_code.return_code
    when (parameter = 'buffer') then
      say '  buffer is' left(buffer,send_length)
    when (parameter = 'receive_buffer') then
      say '  buffer is' left(receive_buffer,received_length)
    when (parameter = 'data_received') then
      say '  data_received is' cm_data_received.data_received
    when (parameter = 'status_received') then
      say '  status_received is' cm_status_received.status_received
    when (parameter = 'request_to_send_received') then
      say '  request_to_send_received is',
          cm_request_to_send_received.request_to_send_received
    when (parameter = 'send_type') then
      say '  send_type is' cm_send_type.send_type
    when (parameter = 'deallocate_type') then
      say '  deallocate_type is' cm_deallocate_type.deallocate_type
```

```
    when (parameter = 'conversation_type') then
      say '  conversation_type is',
          cm_conversation_type.conversation_type
    otherwise
      say ' ' parameter 'is' value(parameter)
  end
end
/*--------------------------------------------------------------------*/
/* Extract the current conversation state of the local program.       */
/*--------------------------------------------------------------------*/
/* Commenting out next four lines ...
'CMECS conversation_ID conversation_state return_code'
if (return_code = CM_OK) then
  say '  conversation_state is =>',
      cm_conversation_state.conversation_state
... */

return


Error:
/*--------------------------------------------------------------------*/
/* Report error when REXX special variable RC is not 0.               */
/*--------------------------------------------------------------------*/
say
say '* ERROR:  REXX has detected an error'
say '          The return code variable RC was set to' rc
call AbnormalEnd
signal GetOut


ErrorHandler:
/*--------------------------------------------------------------------*/
/* Report routine that failed and the error return code.              */
/*--------------------------------------------------------------------*/
parse arg routine_name
say
say '* ERROR:  An error occurred during a' routine_name 'call'
say '          The return_code was set to' cm_return_code.return_code
call AbnormalEnd
signal GetOut

AbnormalEnd:
/*--------------------------------------------------------------------*/
/* Abnormally deallocate the conversation.  Since we are exiting      */
/* due to an error, we will not display an error message if the       */
/* Set_Deallocate_Type or Deallocate call encounters an error.        */
/*--------------------------------------------------------------------*/
deallocate_type = CM_DEALLOCATE_ABEND
'CMSDT conversation_ID deallocate_type return_code'
say; say 'Routine called:  CMSDT'
if (return_code = CM_OK) then
  do
    call TraceParms
    'CMDEAL conversation_ID return_code'
    say; say 'Routine called:  CMDEAL'
    if (return_code = CM_OK) then
      call TraceParms
  end

return
```

Summary

You have reached the end of the advanced-function calls chapter. In this chapter, we learned more about conversation states. We explored confirmation processing and the setting and extracting of conversation characteristics. Along the way, we discovered how conversation characteristics can influence the course of a conversation. It should be clear that the SAA CPI Communications calls we have covered provide powerful function.

Next, we will examine some of the VM extension calls to CPI Communications. We will be using the same execs in the next section, so do not erase them. You might want to make a backup copy of the execs at this point.

Chapter 4. VM Extensions to CPI Communications

Now that we are familiar with most of the SAA CPI Communications routines, let's direct our attention to the VM extensions. These extension routines let you take advantage of VM's ability to provide additional security levels, to accept multiple conversations for each transaction program (TP), and to set up an intermediate server.

The Relationship between VM and SAA CPI Communications

Before looking at the VM extension calls and what we can do with them, let's step back for a moment and briefly discuss the communications functions and protocols underlying SAA CPI Communications. This will help us to understand VM's implementation of and extensions to CPI Communications.

SAA CPI Communications provides a programming interface to IBM's Systems Network Architecture (SNA) logical unit 6.2 (LU 6.2). SNA LU 6.2 defines a set of communications functions and protocols that let application programs on different systems communicate. The set of calls defined by SAA, however, does not implement every aspect of the LU 6.2 protocol. VM provides extensions to SAA CPI Communications to support several additional LU 6.2 features, such as support for security types. VM also provides routines that can be considered extensions to the LU 6.2 architecture. Resource manager support for accepting multiple incoming conversations, for example, is not part of the LU 6.2 protocol. This support is very important to server programs in the VM environment.

CPI Communications applications running in the VM environment can establish conversations that closely conform to the LU 6.2 model for communications. Such applications are referred to here as LU 6.2 transaction program model applications, or *TP-model applications*. While a TP-model application can be created using only SAA CPI Communications routines, such an application is also allowed to call most of the VM/ESA extension routines.

For more detailed information on TP-model applications, see the "Understanding CPI Communications" chapter in the *z/VM: CMS Application Development Guide*.

Overview of VM Extension Calls

VM provides extension routines for setting and extracting access security information and for supporting multiple concurrent conversations per server. These functions do not exist in SAA CPI Communications.

Note: After VM extensions are added to a program, the program will require modification if it is ported to another SAA platform. The VM extensions can be used with the SAA starter set and advanced-function calls, but it may be beneficial to place them in separate procedures whenever possible to facilitate modifying your program in case you need to port it to another SAA platform later.

We will divide this chapter into sections that group the VM extension routines according to their use in our programs. First, we will discuss resource managers, followed by some general security considerations. Then, we will discuss intermediate servers and the security concerns relating specifically to them.

Summary of VM Extension Calls

You can identify the extension calls easily because they all begin with the prefix XC. In VM, these routines can be logically divided into several categories, as the following tables show. Some of the routines are not discussed because they are beyond the scope of this book.

Calls Used for Conversation Security

Pseudonym	Call	Description	Page
Extract_Conversation_Security_User_ID	XCECSU	Extracts the access security user ID for the conversation	184
Set_Client_Security_User_ID	XCSCUI	Lets an intermediate server specify a client user ID	191
Set_Conversation_Security_Password	XCSCSP	Sets the access security password for the conversation	162
Set_Conversation_Security_Type	XCSCST	Sets the security level for the conversation	158
Set_Conversation_Security_User_ID	XCSCSU	Sets the access security user ID for the conversation	161

Calls Used for Resource Management and Event Notification

Pseudonym	Call	Description	Page
Identify_Resource_Manager	XCIDRM	Identifies a name for a given resource to be managed by this program (resource manager)	135
Signal_User_Event	XCSUE	Queues an event to be reported by a subsequent Wait_on_Event call in the same virtual machine	198
Terminate_Resource_Manager	XCTRRM	Ends ownership of a resource by a resource manager	139
Wait_on_Event	XCWOE	Allows an application to wait on an event from one or more partners (events that can be reported include user events, allocation events, information input, notification that a resource has been revoked, console input, and Shared File System asynchronous events)	143

Calls Used for Resource Recovery Support

Pseudonym	Call	Description	Page
Extract_Conversation_LUWID	XCECL	Extracts the logical unit of work ID associated with the specified protected conversation	198
Extract_Local_Fully_Qualified_LU_Name	XCELFQ	Extracts the local fully-qualified LU name for the specified conversation	198
Extract_Remote_Fully_Qualified_LU_Name	XCERFQ	Extracts the remote fully-qualified LU name for the specified conversation	198
Extract_TP_Name	XCETPN	Extracts the TP name for the specified conversation	198

Call Used for Extracting CMS Work Unit ID

Pseudonym	Call	Description	Page
Extract_Conversation_Workunitid	XCECWU	Extracts the CMS work unit ID associated with the specified conversation	198

Managing a Resource

In this section, we will use VM extension routines to convert our server program into a resource manager that can handle requests for a particular resource (in this case, a file called TEST FILE).

What Is a Resource Manager?

A resource manager is simply a program that controls access to a resource, such as a file, database, device, or other entity that can be identified for application program processing.

A resource is identified in VM by a name called a resource ID. When a user requests a connection to a specific resource, the resource manager program handles the request.

By identifying itself as a resource manager, a server application can manage one or more resources and can accept more than one conversation per resource. Our purpose here is to acquaint you with the concepts and the calls to implement them, so we will not actually demonstrate the management of more than one resource or the acceptance of more than one conversation.

What Kinds of Resources Are There?

We briefly discussed resources in the context of communications programming earlier in the book, but let's review and expand upon what we know about them. We mentioned four kinds of resources: local, global, system, and private.

Local

A local resource is known only to the local VM system. Only authorized users on the local system can access the resource. In addition, the names of the local resources must be unique within the system where they reside.

A local resource manager must be logged on and already running before users can make a successful allocate conversation request.

Resources (for example, a printer) that should be limited to the users of one system should be defined as a local resource to that system.

Global

A global resource is known to the local system, to all systems within the TSAF or CS collection and to the SNA network. Global resource names must be unique within the TSAF or CS collection. Authorized users in the TSAF or CS collection or in the SNA network can access global resources.

A global resource manager must be logged on and already running before users can make a successful allocate conversation request.

System

A system resource is known only to the VM/ESA system where it is located but is remotely accessible from other systems. A system resource name only needs to be unique to that system. Any authorized user in the TSAF or CS collection or the SNA network connected to the system on which the system resource resides can access the system resource.

A system resource manager must be logged on and already running before users can make a successful allocate conversation request.

Private

A private resource is known only to the virtual machine where it resides. As we have seen, the $SERVER$ NAMES file identifies the various private resources and validates all allocation requests. Private resource names need to be unique only within the virtual machine where they reside. Any authorized users in the TSAF or CS collection or in the SNA network can access private resources.

A private resource manager need not be logged on when the allocation request is presented. CP will autolog a private resource manager and automatically invoke the TP associated with a given resource found in the $SERVER$ NAMES file.

Resources that need to be limited to a single user should be defined as private. For example, a user working on a workstation uses a program to access files maintained in a virtual machine. These files would be defined as private resources and the workstation user would be the only authorized user of the resources.

Using VM Extension Calls to Manage Resources

The following table shows in pseudocode style how our two programs are changing. The calls we will be adding in this section appear in boldface.

Table 4. Overview of Sample Programs Using VM Extensions

REQUESTR User ID	SERVR User ID

Initialize_Conversation
Set_Conversation_Security_Type
Set_Conversation_Security_User_ID
Set_Conversation_Security_Password
Set_Partner_LU_Na,me
Set_TP_Na,me
Allocate
Send_Data
if performing confirmation
  Confirm
Set_Prepare_To_Receive_Type
Prepare_To_Receive
 	
-Receive loop-
do until send control returned
  Receive
  if confirmation requested
  Confirmed
end
 	
Deallocate
	Identify_Resource_Manager
	Do forever
	  Wait_on_Event
	  select on event type
	  when allocation request
	  Accept_Conversation
	  Extract_Conversation_Type
	  if conversation type is basic
	  Send_Error
	  when information input
	
	  -Receive loop-
	  do until no data left to receive
	  Receive
	  if confirmation requested
	  Confirmed
	  if send control received
	
	  -Send loop-
	  do until whole file is sent
	  if last data record
	  Set_Send_Type

Although we are not adding many new calls to our programs, you will notice that we are restructuring SENDBACK EXEC on the SERVR user ID. This restructuring will make our server program more flexible.

FYI: Tidying Up, Part III

In case you have not been removing unneeded parameters from the TraceParms subroutine calls, you might want to go back and clean up the PROCESS and SENDBACK execs now. Remove all the parameters on the TraceParms subroutine calls that do not provide you with useful information (only keep the *status_received* and *data_received* parameters). Remember, we will be adding more of these calls for all the routines introduced in this chapter and we do not want the console log to be too long.

If you just deleted the parameters on the TraceParms subroutine calls, rerun the execs to make sure they complete successfully. We will not be using confirmation processing in the rest of the program, so you can either remember to answer "N" to the confirmation-processing prompt each time you run the program, or you can comment out the following line in the PROCESS EXEC

```
parse upper pull perform_confirm
```

and add this declaration line to make the choice automatic:

```
perform_confirm = 'N'
```

The Identify_Resource_Manager (XCIDRM) Call

A server program uses the Identify_Resource_Manager (XCIDRM) call to declare to CMS the name of the resource that it wants to manage.

An application can call Identify_Resource_Manager multiple times to identify different resources that it wants to manage.

The format for Identify_Resource_Manager is:

```
CALL  XCIDRM(resource_ID,                    input
             resource_manager_type,          input
             service_mode,                   input
             security_level_flag,            input
             return_code)                    output
```

Input Parameters

The Identify_Resource_Manager call expects four input parameters, the first of which is the ***resource_ID***, which specifies the name of a resource to be managed by this resource manager application. The *resource_ID* parameter value corresponds to the transaction program name provided by applications that allocate a conversation for this resource. Those allocation requests are then routed to the application that called this Identify_Resource_Manager routine.

Use the ***resource_manager_type*** parameter to specify whether the *resource_ID* contains the name of a private, local, global, or system resource. Valid values are:

XC_PRIVATE (0)
: indicates that the specified resource will be managed as a private resource, accessible to authorized users residing in the local system, a TSAF or CS collection, or an SNA network.

XC_LOCAL (1)
: indicates that the specified resource will be managed as a local resource, accessible only to users within the local system.

XC_GLOBAL (2)
: indicates that the specified resource will be managed as a global resource, accessible to users in the local system, the TSAF or CS collection, or the SNA network.

XC_SYSTEM (3)
: indicates that the specified resource will be managed as a system resource, accessible to users in the local system, the TSAF or CS collection, or the SNA network.

Use the ***service_mode*** parameter to specify how this resource manager application handles conversations. Valid values are:

XC_SINGLE (0)
: indicates that this resource manager program can accept only a single conversation for the specified *resource_ID*.

  After the resource manager program has accepted the single conversation, further allocation requests for the same *resource_ID* will be queued for private resources and deallocated for local and global resources.

  After the resource manager program has completed processing and deallocated the single conversation, it should call the Terminate_Resource_Manager (XCTRRM) routine before ending. A private resource manager application that has queued allocation requests pending will be restarted as soon as it ends.

XC_SEQUENTIAL (1)
: indicates that this resource manager program can accept only one conversation at a time for a given *resource_ID*.

  After the resource manager program has accepted a conversation, further allocation requests for the same *resource_ID* will be queued for private resources and deallocated for local and global resources.

  When a conversation is completed and deallocated, the resource manager program can issue Wait_on_Event (XCWOE) to wait for the next allocation request.

XC_MULTIPLE (2)
: indicates that this resource manager program can accept multiple concurrent conversations for a given *resource_ID*.

  Subsequent allocation requests for the same *resource_ID* will be presented to the resource manager program the next time Wait_on_Event (XCWOE) is called.

Use the ***security_level_flag*** parameter to specify whether this resource manager will accept allocation requests with *conversation_security_type* set to XC_SECURITY_NONE (which is identical to having a `:security.` tag value of NONE in the corresponding communications directory entry). Valid values are:

- XC_REJECT_SECURITY_NONE (0)
- XC_ACCEPT_SECURITY_NONE (1)

Output Parameter

Possible values for the ***return_code*** parameter are:

CM_OK (0)
: indicates that the Identify_Resource_Manager (XCIDRM) call completed successfully.

CM_PRODUCT_SPECIFIC_ERROR (20)
: indicates a CMS error; check the CPICOMM LOGDATA file.

CM_PROGRAM_PARAMETER_CHECK (24)
: indicates that the *resource_ID* has already been defined within the virtual machine or that an invalid value was specified for the *resource_manager_type*, *service_mode*, or *security_level_flag*.

(25)
: indicates that this is a TP-model application, so Identify_Resource_Manager (XCIDRM) cannot be called. See the discussion of TP-model applications "The Relationship between VM and SAA CPI Communications" on page 131 for more information.

CM_UNSUCCESSFUL (28)
: indicates that the *resource_ID* is already defined for another virtual machine or that you do not have VM directory authorization to identify this local or global resource.

Results of the Call

When *return_code* indicates CM_OK, the resource ID has been declared. This call does not cause a state change.

Adding XCIDRM to Our Server Program

The server application that we have created can be considered a private resource manager. Although this call is not required for our program and does not change the results, it will show how to set up a resource manager. As an exercise, you may want to try extending this program to accept multiple conversations for the same resource.

We will add a call to Identify_Resource_Manager immediately preceding the Accept_Conversation (CMACCP) call. Recall that the name of the private resource being requested is passed to the resource manager as an argument. This private resource name also happens to be the value specified on the `:nick.` tag in $SERVER$ NAMES, as illustrated in Figure 12 on page 32. We will use that value passed as an argument to identify our resource on the Identify_Resource_Manager call. In our case, it will be GET. Note that we do not have to use the value passed as an argument for this purpose; we could identify the resource to be managed simply by specifying its name in the program. The *resource_manager_type* will be XC_PRIVATE. We have been dealing with just one conversation, so XC_SINGLE will suffice for the *service_mode*, and we will exclude connections having a *conversation_security_type* of XC_SECURITY_NONE by setting the *security_level_flag* to XC_REJECT_SECURITY_NONE.

Because there is no *conversation_ID* associated with the Identify_Resource_Manager call, the ErrorHandler subroutine will not need to be called if an error is detected. We will just have the program display an error message.

The SENDBACK EXEC should now have the following lines:

```
/*====================================================================*/
/*  SENDBACK EXEC - Sample server application.                        */
/*====================================================================*/

arg resource_ID .          /* :nick. value from $SERVER$ NAMES file */
/*--------------------------------------------------------------------*/
/* Set up REXX environment for program-to-program communications    */
/* and enable error trapping of REXX errors.                         */
/*--------------------------------------------------------------------*/
address cpicomm
signal on error
/*--------------------------------------------------------------------*/
```

```
/* Equate pseudonyms to their integer values based on the          */
/* definitions contained in the CMREXX COPY file.                  */
/*-----------------------------------------------------------------*/
address command 'EXECIO * DISKR CMREXX COPY * (FINIS STEM PSEUDONYM.'
do index = 1 to pseudonym.0
  interpret pseudonym.index
end
/*-----------------------------------------------------------------*/
/* Identify the application as manager of the private resource.    */
/*-----------------------------------------------------------------*/
resource_manager_type = XC_PRIVATE
service_mode = XC_SINGLE
security_level_flag = XC_REJECT_SECURITY_NONE
'XCIDRM resource_ID resource_manager_type service_mode',
       'security_level_flag return_code'
say; say 'Routine called:  XCIDRM'
if (return_code ¬= CM_OK) then
  do
    say
    say '* ERROR:  An error occurred during an XCIDRM call'
    say '          The return_code was set to',
        cm_return_code.return_code
    signal GetOut
  end
call TraceParms 'resource_ID resource_manager_type service_mode',
                'security_level_flag return_code'
/*-----------------------------------------------------------------*/
/* Accept the incoming conversation.                               */
/*-----------------------------------------------------------------*/
'CMACCP conversation_ID return_code'
say; say 'Routine called:  CMACCP'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMACCP'
call TraceParms
  ⋮

/*-------------------------- Subroutines --------------------------*/

  ⋮

TraceParms:
/*-----------------------------------------------------------------*/
/* Display parameters and their values as passed to this subroutine.*/
/*-----------------------------------------------------------------*/
parse arg parmlist
do word_num = 1 to words(parmlist)
  parameter = word(parmlist,word_num)
  select
    when (parameter = 'return_code') then
      say '  return_code is' cm_return_code.return_code
    when (parameter = 'buffer') then
      say '  buffer is' left(buffer,send_length)
    when (parameter = 'receive_buffer') then
      say '  buffer is' left(receive_buffer,received_length)
    when (parameter = 'data_received') then
      say '  data_received is' cm_data_received.data_received
    when (parameter = 'status_received') then
      say '  status_received is' cm_status_received.status_received
    when (parameter = 'request_to_send_received') then
      say '  request_to_send_received is',
          cm_request_to_send_received.request_to_send_received
    when (parameter = 'send_type') then
      say '  send_type is' cm_send_type.send_type
    when (parameter = 'deallocate_type') then
      say '  deallocate_type is' cm_deallocate_type.deallocate_type
    when (parameter = 'conversation_type') then
      say '  conversation_type is',
          cm_conversation_type.conversation_type
```

```
    when (parameter = 'resource_manager_type') then
      say '  resource_manager_type is',
          xc_resource_manager_type.resource_manager_type
    when (parameter = 'service_mode') then
      say '  service_mode is' xc_service_mode.service_mode
    when (parameter = 'security_level_flag') then
      say '  security_level_flag is',
          xc_security_level_flag.security_level_flag
    otherwise
      say ' ' parameter 'is' value(parameter)
  end
end
/*-------------------------------------------------------------------*/
/* Extract the current conversation state of the local program.      */
/*-------------------------------------------------------------------*/
/* Commenting out next four lines ...
'CMECS conversation_ID conversation_state return_code'
if (return_code = CM_OK) then
  say '  conversation_state is =>',
      cm_conversation_state.conversation_state
... */

return
   ⋮
```

An application that calls Identify_Resource_Manager (XCIDRM) to declare ownership of a resource should also call Terminate_Resource_Manager (XCTRRM) to end its ownership of that resource before exiting. Let's examine the Terminate_Resource_Manager call next and add it to our program before trying out our revised program.

The Terminate_Resource_Manager (XCTRRM) Call

The Terminate_Resource_Manager (XCTRRM) call is used by a resource manager application to terminate ownership of a resource. Any conversations and pending allocation requests associated with the specified *resource_ID* will be automatically deallocated.

The format for Terminate_Resource_Manager is:

CALL XCTRRM(*resource_ID,*		input
	return_code)	output

Input Parameter

Use the ***resource_ID*** parameter to specify the resource name, previously designated for management by this resource manager application on a call to Identify_Resource_Manager (XCIDRM), for which service is being terminated.

Output Parameter

Possible values for the ***return_code*** parameter are:

CM_OK (0)
: indicates that Terminate_Resource_Manager completed successfully.

CM_PROGRAM_PARAMETER_CHECK (24)
: indicates that the virtual machine does not control the specified resource.

CM_PROGRAM_STATE_CHECK (25)
: indicates that this is a TP-model application, so Terminate_Resource_Manager cannot be called. See the discussion of TP-model applications "The Relationship between VM and SAA CPI Communications" on page 131 for more information.

Results of the Call

When *return_code* indicates CM_OK, the resource is no longer identified (any future allocates to it will fail) and any conversations associated with the specified *resource_ID* enter **Reset** state.

Adding XCTRRM to Our Server Program

Let's add the Terminate_Resource_Manager call to SENDBACK EXEC in a subroutine called TerminateRes. After the server receives the deallocation notice from its partner, it will call TerminateRes to free up the resource this program has been managing.

We will add a call to TerminateRes following the GetOut label so our program will always terminate ownership of the resource before exiting. Just as we issue Deallocate (CMDEAL) with *deallocate_type* set to CM_DEALLOCATE_ABEND from the ErrorHandler subroutine to try to ensure that the conversation is deallocated by the program, adding the TerminateRes call following the GetOut label will ensure that we terminate ownership of the resource when an error has been detected as well as during normal program termination. Note that all conversations associated with the resource should be deallocated before calling Terminate_Resource_Manager.

Your server program should now have the following lines:

```
/*====================================================================*/
/*  SENDBACK EXEC - Sample server application.                        */
/*====================================================================*/

  ⋮
/*--------------------------------------------------------------------*/
/* Start a Receive loop.                                              */
/* Receive data, status, or both from conversation partner.           */
/*--------------------------------------------------------------------*/
requested_file = ''
requested_length = 20
do until (CMRCV_return_code ¬= CM_OK) |,
         (status_received = CM_CONFIRM_DEALLOC_RECEIVED)
  'CMRCV conversation_ID receive_buffer requested_length',
        'data_received received_length status_received',
        'request_to_send_received return_code'
  CMRCV_return_code = return_code
  say; say 'Routine called:  CMRCV'
  select
    when (CMRCV_return_code = CM_OK) then
      do
  ⋮
      end
    when (CMRCV_return_code = CM_DEALLOCATED_NORMAL) then
      do
        call TraceParms
        say; say 'Conversation deallocated by partner'
      end
    otherwise
      call ErrorHandler 'CMRCV'
  end
end

GetOut:
  call TerminateRes
  exit

/*------------------------- Subroutines --------------------------*/
```

```
   ⋮

AbnormalEnd:
/*-------------------------------------------------------------------*/
/* Abnormally deallocate the conversation.  Since we are exiting     */
/* due to an error, we will not display an error message if the      */
/* Set_Deallocate_Type or Deallocate call encounters an error.       */
/*-------------------------------------------------------------------*/
deallocate_type = CM_DEALLOCATE_ABEND
'CMSDT conversation_ID deallocate_type return_code'
say; say 'Routine called:  CMSDT'
if (return_code = CM_OK) then
  do
    call TraceParms 'conversation_ID deallocate_type return_code'
    'CMDEAL conversation_ID return_code'
    say; say 'Routine called:  CMDEAL'
    if (return_code = CM_OK) then
      call TraceParms
  end

return


TerminateRes:
/*-------------------------------------------------------------------*/
/* TerminateRes will terminate ownership of the specified resource. */
/*-------------------------------------------------------------------*/
'XCTRRM resource_ID return_code'
say; say 'Routine called:  XCTRRM'
if (return_code ¬= CM_OK) then
  do
    say
    say '* ERROR:  An error occurred during an XCTRRM call'
    say '          The return_code was set to',
        cm_return_code.return_code
  end
else
  call TraceParms 'resource_ID return_code'

return
```

Everything should work fine if you try out our latest change. The REQUESTR user ID results should be:

```
process getfile
Requesting the file:  TEST FILE A

Routine called:  CMINIT

Routine called:  CMSPLN

Routine called:  CMSTPN

Would you like confirmation processing? (Y/N)
N

Routine called:  CMALLC

Routine called:  CMSEND

Routine called:  CMSPTR

Routine called:  CMPTR

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_SEND_RECEIVED

Routine called:  CMDEAL
Ready;
```

Figure 49. Results of PROCESS EXEC Execution

The SERVR user ID results should be:

```
Routine called:  XCIDRM
  resource_ID is GET
  resource_manager_type is XC_PRIVATE
  service_mode is XC_SINGLE
  security_level_flag is XC_REJECT_SECURITY_NONE
  return_code is CM_OK

Routine called:  CMACCP

Routine called:  CMECT

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_SEND_RECEIVED

Routine called:  CMSEND

Routine called:  CMSST

Routine called:  CMSEND

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Conversation deallocated by partner

Routine called:  XCTRRM
  resource_ID is GET
  return_code is CM_OK
Ready;
```

Figure 50. SENDBACK EXEC Execution as a Resource Manager

The Wait_on_Event (XCWOE) Call

The Wait_on_Event (XCWOE) call is used by an application to wait for a communications event from one or more partners. In addition, Wait_on_Event can reflect console interrupts and user events to the program.

Wait_on_Event is often used by resource managers that wait for some type of interrupt to determine the next action that is to be performed.

Note: For multitasking applications, it is recommended that you use CMS event management services and the VMCPIC system event, rather than Wait_on_Event. You can find introductory information about using event management services for CPI Communications 217.

The format for Wait_on_Event is:

```
CALL  XCWOE(resource_ID,                output
            conversation_ID,            output
            event_type,                 output
            event_info_length,          output
            event_buffer,               output
            return_code)                output
```

Output Parameters

The ***resource_ID*** parameter returns the name of a resource (managed by the calling resource manager application) for which an event has occurred. The value returned is a name that was specified by this application on a previous Identify_Resource_Manager (XCIDRM) call.

The *resource_ID* parameter contains a meaningful value only when the *event_type* is XC_ALLOCATION_REQUEST, XC_RESOURCE_REVOKED, or XC_USER_EVENT.

The ***conversation_ID*** parameter returns an identifier for the conversation on which information is available to be received. The *conversation_ID* parameter contains a meaningful value only when the *event_type* is XC_INFORMATION_INPUT.

The ***event_type*** parameter returns a value indicating the type of event that has occurred. The *event_type* can be set to one of the following values:

XC_ALLOCATION_REQUEST (1)
: indicates that a remote program is attempting to allocate a conversation to this application. This event will continue to be reported by subsequent calls to Wait_on_Event until you issue an Accept_Conversation (CMACCP) call to process the event or until Terminate_Resource_Manager (XCTRRM) is called to end management of the subject resource.

XC_INFORMATION_INPUT (2)
: indicates that a communications partner is attempting to send data, status, or both to this application. This event will continue to be reported by subsequent calls to Wait_on_Event until you issue a Receive (CMRCV) call to process this event, until a Send_Error (CMSERR) or Deallocate (CMDEAL) call is issued, or until Terminate_Resource_Manager is called to end management of the subject resource.

XC_RESOURCE_REVOKED (3)
: indicates that another program has revoked the resource being managed by this resource manager application. The resource manager application must issue a Terminate_Resource_Manager call when it completes all active conversations. After it has been presented to the application, the information associated with this event will no longer be available.

XC_CONSOLE_INPUT (4)
: indicates that information was entered at the console attached to this virtual machine and was placed in the *event_buffer* parameter. After it has been presented to the application, the information associated with this event will no longer be available.

XC_REQUEST_ID (5)
: indicates that a Shared File System asynchronous event has occurred. The request ID is placed in the *event_info_length* parameter. After it has been presented to the application, the information associated with this event will no longer be available.

XC_USER_EVENT (6)
: indicates the occurrence in the caller's virtual machine of some event that is of interest to the program calling Wait_on_Event. The event was detected by another program, such as an interrupt handler, which in turn called the Signal_User_Event (XCSUE) routine to queue the event for reporting by Wait_on_Event. After it has been presented to the application, the information associated with this event will no longer be available.

The ***event_info_length*** parameter returns an integer value with various meanings depending on the particular event type. The *event_info_length* parameter can have one of the following meanings:

- If the *event_type* is XC_INFORMATION_INPUT, XC_CONSOLE_INPUT, or XC_USER_EVENT, *event_info_length* indicates the number of data bytes that are available to be received. When the *event_type* is XC_INFORMATION_INPUT, the value of *event_info_length* should be used on a subsequent Receive call to receive the data. For mapped conversations, this length may be greater than the number of bytes sent by the remote program.
- If the *event_type* is XC_CONSOLE_INPUT or XC_USER_EVENT, *event_info_length* indicates the length of the data that is available in the *event_buffer*.
- If the *event_type* is XC_REQUEST_ID, *event_info_length* contains the actual request ID.

The *event_info_length* parameter does not contain a meaningful value and should not be examined if the *event_type* is XC_ALLOCATION_REQUEST or XC_RESOURCE_REVOKED.

The ***event_buffer*** parameter returns the data, up to 130 bytes, that was either entered at the console during a console input event or passed on a Signal_User_Event call. The *event_buffer* parameter contains a meaningful value only when the *event_type* is XC_CONSOLE_INPUT or XC_USER_EVENT.

Possible values for the ***return_code*** parameter are:

CM_OK (0)
: indicates that the Wait_on_Event call completed successfully.

CM_PRODUCT_SPECIFIC_ERROR (20)
: indicates a CMS error; check the CPICOMM LOGDATA file.

CM_PROGRAM_STATE_CHECK (25)
: indicates that no conversations exist and no resources were identified.

Results of the Call

If multiple events are pending, they will be reported on successive calls to Wait_on_Event (XCWOE) in the following order of priority:

1. User Event
2. Allocation request
3. Information input
4. Resource revoked notification
5. Request ID
6. Console input

So, if a user event occurs, it will be presented on the next Wait_on_Event call ahead of any other event types, even though they may have been pending for longer than the user event.

This call does not cause a state change.

Adding XCWOE to Our Server Program

We will add the Wait_on_Event call to SENDBACK EXEC, following the Identify_Resource_Manager (XCIDRM) call. The only event types our program needs to wait for are allocation requests and information input.

Typically, Wait_on_Event is placed inside a loop, and a resource manager calling Wait_on_Event would generally want to keep its end of a conversation in **Receive** state.

When Wait_on_Event completes, the section of code designed to handle the particular event that was reported gets executed. Then, Wait_on_Event can be issued again to wait for the next event to occur.

We will need to restructure our server program when we add the Wait_on_Event call. In effect, we are going to let our program be driven by the type of event that gets reported. For that reason, we will want to make our program more generic. Instead of adding each routine call to the program in the order that it will be issued, we will group into a subroutine the calls that are related for a particular event.

We will need to move some of our existing code into subroutines. In addition, because the *resource_ID* and *conversation_ID* parameters on the Wait_on_Event call may not always contain meaningful information, we will want to keep the resource name and conversation ID in two additional variables, *save_res_ID* and *save_con_ID*, so that the information is not lost. That way, if an error is detected, the program will always have access to the resource ID and conversation ID so it can terminate management of the resource and deallocate the conversation.

We can also change the Receive (CMRCV) call so that the *requested_length* is set to the value in the Wait_on_Event parameter *event_info_length* when there is information to be received. That way, we will be requesting to receive the amount of data that is available to the program. Keep in mind, however, that an application using this value as the requested length may need to verify that it does not exceed the maximum length for a single Receive call.

To keep the amount of screen output reasonable, we will not display the *event_buffer* with the Wait_on_Event results. But we will continue to display the *resource_ID* associated with both the Identify_Resource_Manager and the Terminate_Resource_Manager calls.

Here is what the modified server program looks like:

```
/*====================================================================*/
/*  SENDBACK EXEC - Sample server application.                        */
/*====================================================================*/

  ⋮
/*--------------------------------------------------------------------*/
/* Equate pseudonyms to their integer values based on the             */
/* definitions contained in the CMREXX COPY file.                     */
/*--------------------------------------------------------------------*/
address command 'EXECIO * DISKR CMREXX COPY * (FINIS STEM PSEUDONYM.'
do index = 1 to pseudonym.0
  interpret pseudonym.index
end
/*--------------------------------------------------------------------*/
/* Identify the application as manager of the private resource.       */
/* Remember the resource_ID value for later use in XCTRRM by          */
/* storing it in save_res_ID.                                         */
/*--------------------------------------------------------------------*/
save_res_ID = resource_ID
resource_manager_type = XC_PRIVATE
service_mode = XC_SINGLE
security_level_flag = XC_REJECT_SECURITY_NONE
'XCIDRM resource_ID resource_manager_type service_mode',
       'security_level_flag return_code'
say; say 'Routine called:  XCIDRM'
if (return_code ¬= CM_OK) then
  do
    say
    say '* ERROR:  An error occurred during an XCIDRM call'
```

```
    say '          The return_code was set to',
        cm_return_code.return_code
    signal GetOut
  end
call TraceParms 'resource_ID'
/*-------------------------------------------------------------------*/
/* Start continuous Wait_on_Event loop.                              */
/*-------------------------------------------------------------------*/
requested_file = ''
do forever
  /*-----------------------------------------------------------------*/
  /* Issue Wait_on_Event to wait for the next event to occur.        */
  /*-----------------------------------------------------------------*/
  'XCWOE resource_ID conversation_ID event_type event_info_length',
         'event_buffer return_code'
  say; say 'Routine called:  XCWOE'
  if (return_code ¬= CM_OK) then call ErrorHandler 'XCWOE'
  call TraceParms 'resource_ID conversation_ID event_type',
                  'event_info_length return_code'
  /*-----------------------------------------------------------------*/
  /* Choose next action based on type of event.                      */
  /*-----------------------------------------------------------------*/
  select
    when (event_type = XC_ALLOCATION_REQUEST) then
      call AcceptConv
    when (event_type = XC_INFORMATION_INPUT) then
      call ReceiveInfo
    otherwise
      do
        say
        say '* ERROR:  Wait_on_Event reported event_type',
            xc_event_type.event_type
      end
  end /* select */
  /*-----------------------------------------------------------------*/
  /* When notice of partner's deallocation is received, leave the    */
  /* Wait_on_Event loop.                                             */
  /*-----------------------------------------------------------------*/
  if (return_code = CM_DEALLOCATED_NORMAL) |,
     (status_received = CM_CONFIRM_DEALLOC_RECEIVED) then
    leave
end /* do forever */

GetOut:
  call TerminateRes
  exit

/*------------------------- Subroutines --------------------------*/

AcceptConv:
/*-------------------------------------------------------------------*/
/* Accept the incoming conversation.                                 */
/* Store the conversation identifier in save_con_ID.                 */
/*-------------------------------------------------------------------*/
'CMACCP conversation_ID return_code'
save_con_ID = conversation_ID
say; say 'Routine called:  CMACCP'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMACCP'
call TraceParms
/*-------------------------------------------------------------------*/
/* Extract conversation_type to ensure the conversation is mapped.  */
/*-------------------------------------------------------------------*/
'CMECT conversation_ID conversation_type return_code'
say; say 'Routine called:  CMECT'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMECT'
call TraceParms
/*-------------------------------------------------------------------*/
```

```
/* If the conversation is basic, deallocate abnormally.             */
/*------------------------------------------------------------------*/
if (conversation_type = CM_BASIC_CONVERSATION) then
  do
    say; say '* ERROR:  Accepting and deallocating a basic',
             'conversation'
    /*----------------------------------------------------------------*/
    /* Call Send_Error to notify partner that error was detected.   */
    /* Since the program is going to exit, do not check the         */
    /* Send_Error results for an error.                             */
    /*----------------------------------------------------------------*/
    'CMSERR conversation_ID request_to_send_received return_code'
    say; say 'Routine called:  CMSERR'
    if (return_code = CM_OK) then
      call TraceParms
    call AbnormalEnd
    signal GetOut
  end

return


ReceiveInfo:
/*------------------------------------------------------------------*/
/* Start a Receive loop.                                            */
/* Receive data, status, or both from conversation partner.         */
/*------------------------------------------------------------------*/
requested_file = ''
requested_length = event_info_length
do until (data_received = CM_COMPLETE_DATA_RECEIVED) |,
         (data_received = CM_NO_DATA_RECEIVED)
  'CMRCV conversation_ID receive_buffer requested_length',
        'data_received received_length status_received',
        'request_to_send_received return_code'
  CMRCV_return_code = return_code
  say; say 'Routine called:  CMRCV'
  select
    when (CMRCV_return_code = CM_OK) then
      do
        call TraceParms 'data_received status_received'
        if (data_received ¬= CM_NO_DATA_RECEIVED) then
          do
            receive_buffer = left(receive_buffer,received_length)
            requested_file = requested_file || receive_buffer
          end
        /*--------------------------------------------------------*/
        /* Determine whether a confirmation request has been      */
        /* received.  If so, respond with a positive reply.       */
        /*--------------------------------------------------------*/
        if (status_received = CM_CONFIRM_RECEIVED) |,
           (status_received = CM_CONFIRM_SEND_RECEIVED) |,
           (status_received = CM_CONFIRM_DEALLOC_RECEIVED) then
          do
            /*------------------------------------------------*/
            /* Issue Confirmed to reply to the partner.       */
            /*------------------------------------------------*/
            'CMCFMD conversation_ID return_code'
            say; say 'Routine called:  CMCFMD'
            if (return_code ¬= CM_OK) then call ErrorHandler 'CMCFMD'
            call TraceParms
          end
        if (status_received = CM_SEND_RECEIVED) |,
           (status_received = CM_CONFIRM_SEND_RECEIVED) then
          call SendFile
        else
          if (status_received = CM_CONFIRM_DEALLOC_RECEIVED) then
            do
```

```
              say; say 'Conversation deallocated by partner'
            end
      end
    when (CMRCV_return_code = CM_DEALLOCATED_NORMAL) then
      do
        call TraceParms 'data_received status_received'
        say; say 'Conversation deallocated by partner'
      end
    otherwise
      call ErrorHandler 'CMRCV'
  end
end

return


SendFile:
/*--------------------------------------------------------------------*/
/* Read the contents of the requested file and send each line of    */
/* the file to the partner program.                                   */
/*--------------------------------------------------------------------*/
address command 'EXECIO * DISKR' requested_file '(FINIS STEM LINE.'
do index = 1 to line.0
  if (index = line.0) then
    /*------------------------------------------------------------------*/
    /* Reset the send_type conversation characteristic just         */
    /* before the final Send_Data call.                                 */
    /*------------------------------------------------------------------*/
    do
      send_type = CM_SEND_AND_PREP_TO_RECEIVE
      'CMSST conversation_ID send_type return_code'
      say; say 'Routine called:  CMSST'
      if (return_code ¬= CM_OK) then call ErrorHandler 'CMSST'
      call TraceParms
    end
  buffer = line.index
  send_length = length(buffer)
  'CMSEND conversation_ID buffer send_length',
         'request_to_send_received return_code'
  say; say 'Routine called:  CMSEND'
  if (return_code ¬= CM_OK) then call ErrorHandler 'CMSEND'
  call TraceParms
end

return


TraceParms:
/*--------------------------------------------------------------------*/
/* Display parameters and their values as passed to this subroutine.*/
/*--------------------------------------------------------------------*/
parse arg parmlist
do word_num = 1 to words(parmlist)
  parameter = word(parmlist,word_num)
  select
    when (parameter = 'return_code') then
      say '  return_code is' cm_return_code.return_code
    when (parameter = 'buffer') then
      say '  buffer is' left(buffer,send_length)
    when (parameter = 'receive_buffer') then
      say '  buffer is' left(receive_buffer,received_length)
    when (parameter = 'data_received') then
      say '  data_received is' cm_data_received.data_received
    when (parameter = 'status_received') then
      say '  status_received is' cm_status_received.status_received
    when (parameter = 'request_to_send_received') then
      say '  request_to_send_received is',
```

```
          cm_request_to_send_received.request_to_send_received
      when (parameter = 'send_type') then
        say '  send_type is' cm_send_type.send_type
      when (parameter = 'deallocate_type') then
        say '  deallocate_type is' cm_deallocate_type.deallocate_type
      when (parameter = 'conversation_type') then
        say '  conversation_type is',
            cm_conversation_type.conversation_type
      when (parameter = 'resource_manager_type') then
        say '  resource_manager_type is',
            xc_resource_manager_type.resource_manager_type
      when (parameter = 'service_mode') then
        say '  service_mode is' xc_service_mode.service_mode
      when (parameter = 'security_level_flag') then
        say '  security_level_flag is',
            xc_security_level_flag.security_level_flag
      when (parameter = 'event_type') then
        say '  event_type is' xc_event_type.event_type
      otherwise
        say ' ' parameter 'is' value(parameter)
    end
end
/*--------------------------------------------------------------------*/
/* Extract the current conversation state of the local program.       */
/*--------------------------------------------------------------------*/
/* Commenting out next four lines ...
'CMECS conversation_ID conversation_state return_code'
if (return_code = CM_OK) then
  say '  conversation_state is =>',
      cm_conversation_state.conversation_state
... */

return

  .
  .
  .

AbnormalEnd:
/*--------------------------------------------------------------------*/
/* Abnormally deallocate the conversation.  Since we are exiting      */
/* due to an error, we will not display an error message if the       */
/* Set_Deallocate_Type or Deallocate call encounters an error.        */
/* Use conversation ID in save_con_ID, from start of conversation.    */
/*--------------------------------------------------------------------*/
deallocate_type = CM_DEALLOCATE_ABEND
conversation_ID = save_con_ID
'CMSDT conversation_ID deallocate_type return_code'
say; say 'Routine called:  CMSDT'
if (return_code = CM_OK) then
  do
    call TraceParms
    'CMDEAL conversation_ID return_code'
    say; say 'Routine called:  CMDEAL'
    if (return_code = CM_OK) then
      call TraceParms
  end

return


TerminateRes:
/*--------------------------------------------------------------------*/
/* TerminateRes will terminate ownership of the specified resource.   */
/* Use resource name stored in save_res_ID at start of program.       */
/*--------------------------------------------------------------------*/
resource_ID = save_res_ID
'XCTRRM resource_ID return_code'
say; say 'Routine called:  XCTRRM'
```

```
if (return_code ¬= CM_OK) then
  do
    say
    say '* ERROR:  An error occurred during an XCTRRM call'
    say '          The return_code was set to',
        cm_return_code.return_code
  end
else
  call TraceParms 'resource_ID'

return
```

Executing the programs now will show some slightly different output.

The following lines should be displayed at the requester's terminal:

```
process getfile
Requesting the file:  TEST FILE A

Routine called:  CMINIT

Routine called:  CMSPLN

Routine called:  CMSTPN

Would you like confirmation processing? (Y/N)
N

Routine called:  CMALLC

Routine called:  CMSEND

Routine called:  CMSPTR

Routine called:  CMPTR

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_SEND_RECEIVED

Routine called:  CMDEAL
Ready;
```

Figure 51. Results of PROCESS EXEC Execution

These lines should appear at the server's terminal:

```
Routine called:  XCIDRM
  resource_ID is GET

Routine called:  XCWOE
  resource_ID is GET
  conversation_ID is
  event_type is XC_ALLOCATION_REQUEST
  event_info_length is 0
  return_code is CM_OK

Routine called:  CMACCP

Routine called:  CMECT

Routine called:  XCWOE
  resource_ID is
  conversation_ID is 00000000
  event_type is XC_INFORMATION_INPUT
  event_info_length is 15
  return_code is CM_OK

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Routine called:  XCWOE
  resource_ID is
  conversation_ID is 00000000
  event_type is XC_INFORMATION_INPUT
  event_info_length is 0
  return_code is CM_OK

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_SEND_RECEIVEDRoutine called:  CMSEND

Routine called:  CMSST

Routine called:  CMSEND

Routine called:  XCWOE
  resource_ID is
  conversation_ID is 00000000
  event_type is XC_INFORMATION_INPUT
  event_info_length is 0
  return_code is CM_OK

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Conversation deallocated by partner

Routine called:  XCTRRM
  resource_ID is GET
Ready;
```

Figure 52. Results of XCWOE to SENDBACK EXEC

Notice that the first Wait_on_Event call completed with an allocation request for the GET resource. Our resource manager could be managing more than just one resource. In that case, the *resource_ID* parameter would let the program know for which resource the request was intended.

After the resource manager accepts a conversation, information input events can occur. These events identify the particular conversation on which the information is available. There being only one conversation involved in our example, the *conversation_ID* is always the same here.

Now that we are convinced the converted exec is working like the previous version of our program, let's try another option for the resource manager. Suppose there were going to be frequent requests to the server for file contents. It might be preferable for the server program to keep running and waiting for these requests rather than terminating after each request is processed.

By identifying our resource with a *service_mode* of XC_SEQUENTIAL, the next allocation request will be reported on a Wait_on_Event (XCWOE) call after the previous conversation has been deallocated.

Our server needs just a couple of changes. We will not be terminating the resource manager when the partner deallocates, so let's add support for console input events to provide a way of stopping the program. That way, the server will continue to run until an entry is made from the SERVR virtual machine's console. Let's also comment out the `LEAVE` statement that was to be performed when an information event resulted in a *return_code* of CM_DEALLOCATED_NORMAL or a *status_received* value of CM_CONFIRM_DEALLOC_RECEIVED.

In addition, we will change the call to the TraceParms subroutine associated with the Wait_on_Event call so that only the *conversation_ID* and *event_type* are displayed. We will also drop *resource_ID* from the Identify_Resource_Manager and Terminate_Resource_Manager TraceParms calls.

Here are the changes to the private resource manager:

```
/*====================================================================*/
/*  SENDBACK EXEC - Sample server application.                        */
/*====================================================================*/

   ⋮
/*--------------------------------------------------------------------*/
/* Equate pseudonyms to their integer values based on the             */
/* definitions contained in the CMREXX COPY file.                     */
/*--------------------------------------------------------------------*/
address command 'EXECIO * DISKR CMREXX COPY * (FINIS STEM PSEUDONYM.'
do index = 1 to pseudonym.0
  interpret pseudonym.index
end
/*--------------------------------------------------------------------*/
/* Identify the application as manager of the private resource.       */
/* Remember the resource_ID value for later use in XCTRRM by          */
/* storing it in save_res_ID.                                         */
/*--------------------------------------------------------------------*/
resource_ID = word(resource_ID 'GET',1)
save_res_ID = resource_ID
resource_manager_type = XC_PRIVATE
service_mode = XC_SEQUENTIAL
security_level_flag = XC_REJECT_SECURITY_NONE
'XCIDRM resource_ID resource_manager_type service_mode',
       'security_level_flag return_code'
say; say 'Routine called:  XCIDRM'
if (return_code ¬= CM_OK) then
  do
    say
    say '* ERROR:  An error occurred during an XCIDRM call'
    say '          The return_code was set to',
        cm_return_code.return_code
```

```
    signal GetOut
  end
call TraceParms
/*--------------------------------------------------------------------*/
/* Start continuous Wait_on_Event loop.                               */
/* Any console input will end the loop.                               */
/*--------------------------------------------------------------------*/
requested_file = ''
do forever
  say; say 'Waiting for an event to occur.  Enter "QUIT" to exit.'
  /*------------------------------------------------------------------*/
  /* Issue Wait_on_Event to wait for the next event to occur.         */
  /*------------------------------------------------------------------*/
  'XCWOE resource_ID conversation_ID event_type event_info_length',
        'event_buffer return_code'
  say; say 'Routine called:  XCWOE'
  if (return_code ¬= CM_OK) then call ErrorHandler 'XCWOE'
  call TraceParms 'conversation_ID event_type'
  /*------------------------------------------------------------------*/
  /* Choose next action based on type of event.                       */
  /*------------------------------------------------------------------*/
  select
    when (event_type = XC_ALLOCATION_REQUEST) then
      call AcceptConv
    when (event_type = XC_INFORMATION_INPUT) then
      call ReceiveInfo
    when (event_type = XC_CONSOLE_INPUT) then
      /*----------------------------------------------------------*/
      /* Leave the Wait_on_Event loop.                            */
      /*----------------------------------------------------------*/
      leave
    otherwise
      do
        say
        say '* ERROR:  Wait_on_Event reported event_type',
            xc_event_type.event_type
      end
  end /* select */
  /*------------------------------------------------------------------*/
  /* When notice of partner's deallocation is received, leave the     */
  /* Wait_on_Event loop.                                              */
  /*------------------------------------------------------------------*/
  /* Commenting out next three lines ...
  if (return_code = CM_DEALLOCATED_NORMAL) |,
     (status_received = CM_CONFIRM_DEALLOC_RECEIVED) then
    leave
  ... */
end /* do forever */

GetOut:
  call TerminateRes
  exit

/*------------------------- Subroutines --------------------------*/

   ⋮

ReceiveInfo:
/*--------------------------------------------------------------------*/
/* Start a Receive loop.                                              */
/* Receive data, status, or both from conversation partner.           */
/*--------------------------------------------------------------------*/
requested_length = event_info_length
do until (data_received = CM_COMPLETE_DATA_RECEIVED) |,
         (data_received = CM_NO_DATA_RECEIVED)
  'CMRCV conversation_ID receive_buffer requested_length',
        'data_received received_length status_received',
```

```
      'request_to_send_received return_code'
  CMRCV_return_code = return_code
  say; say 'Routine called:  CMRCV'
  select
    when (CMRCV_return_code = CM_OK) then
      do
        call TraceParms 'data_received status_received'
        if (data_received ¬= CM_NO_DATA_RECEIVED) then
          do
            receive_buffer = left(receive_buffer,received_length)
            requested_file = requested_file || receive_buffer
          end
        /*-------------------------------------------------------------*/
        /* Determine whether a confirmation request has been           */
        /* received.  If so, respond with a positive reply.            */
        /*-------------------------------------------------------------*/
        if (status_received = CM_CONFIRM_RECEIVED) |,
           (status_received = CM_CONFIRM_SEND_RECEIVED) |,
           (status_received = CM_CONFIRM_DEALLOC_RECEIVED) then
          do
            /*---------------------------------------------------------*/
            /* Issue Confirmed to reply to the partner.                */
            /*---------------------------------------------------------*/
            'CMCFMD conversation_ID return_code'
            say; say 'Routine called:  CMCFMD'
            if (return_code ¬= CM_OK) then call ErrorHandler 'CMCFMD'
            call TraceParms
          end
        if (status_received = CM_SEND_RECEIVED) |,
           (status_received = CM_CONFIRM_SEND_RECEIVED) then
          call SendFile
        else
          if (status_received = CM_CONFIRM_DEALLOC_RECEIVED) then
            do
              say; say 'Conversation deallocated by partner'
              requested_file = ''
            end
      end
    when (CMRCV_return_code = CM_DEALLOCATED_NORMAL) then
      do
        call TraceParms 'data_received status_received'
        say; say 'Conversation deallocated by partner'
        requested_file = ''
      end
    otherwise
      call ErrorHandler 'CMRCV'
  end
end

return
```

⋮

Now, in addition to starting the programs the usual way from the requester virtual machine, you can independently start the resource manager. Recall that the first change we made to SENDBACK in this section was to add the line

```
resource_ID = word(resource_ID 'GET',1)
```

before the Identify_Resource_Manager call. This line lets SENDBACK use the resource ID passed to it if it is started as a result of an allocation request from PROCESS EXEC or, if it is started from the SERVR console, supply the resource ID itself. Let's try the latter case now. From the SERVR user ID, enter

```
sendback
```

You should see:

```
sendback

Routine called:  XCIDRM

Waiting for an event to occur.  Enter "QUIT" to exit.
```

Figure 53. Results of Starting SENDBACK EXEC on the SERVR User ID

The server program will wait until an allocation request or until QUIT (or anything else, for that matter) is entered from the server virtual machine's console. Now start the requester program, as usual, with

```
process getfile
```

The results should be the same as the last time we ran the program, except that the server does not end when the requester program issues Deallocate (CMDEAL).

You can start the requester exec again, and its allocation request will be handled correctly by the server. Try it a few times, and when you would like to terminate the resource manager, simply enter `QUIT` at the console for the server virtual machine. You will then see:

```
QUIT

Routine called:  XCWOE
  conversation_ID is
  event_type is XC_CONSOLE_INPUT

Routine called:  XCTRRM
Ready;
```

Figure 54. Results of Entering QUIT at the SERVR Console

The resource manager terminated ownership of the GET resource and ended. As already mentioned, you can still start the programs from the requester virtual machine, but the server program will no longer terminate by itself. Rather, it will continue waiting for an event to occur.

Security Considerations

When we talk about security in this book, we are referring to access security information for a conversation. This conversation security information includes a user ID and possibly a password that can be sent to the remote LU (only the user ID becomes available to the remote application). This security information verifies the identity of the partner allocating the conversation. The allocating program must supply adequate security information, which is verified by the receiving LU.

SAA CPI Communications does not address the issue of conversation security except to establish a default level of security known as SAME. This means that only the access security ID (user ID) that was used to invoke the local program is sent to the remote LU and transaction program. In addition to conversation security SAME, VM provides NONE and PGM.

SECURITY(NONE)
Indicates that the local program is not sending any access security

information, nor is CP providing a user ID on the allocation request. No user ID information is forwarded, so the remote program cannot determine if the allocation request is authorized.

A public bulletin board would be an example of an application where access authorization is not required.

SECURITY(SAME)
Indicates that the local program is sending neither a user ID nor a password on its allocation. In VM, the user ID that invoked the local program is sent to the remote LU for validation. The remote LU uses the $SERVER$ NAMES file to validate the authorization of the allocation request.

While SAA CPI Communications establishes SECURITY(SAME) as the default security level, it provides no way to examine or change the security level.

SECURITY(SAME) should be used when a program requests services for another program. We will be using SECURITY(SAME) when we discuss the routines that pertain to intermediate servers.

SECURITY(PGM)
Indicates that the local program is sending a user ID and password in its allocation request to access a defined resource for a given conversation. When setting the access security information, any user ID and password combination that is valid at the target LU or within the TSAF collection can be specified. The remote LU is responsible for validating both the user ID and password prior to accepting the allocation request. The $SERVER$ NAMES file contains a list of authorized user IDs for a given private resource. Only the user ID is made available to the remote program.

Side information as defined by SAA CPI Communications does not include any reference to conversation security. VM provides for additional tags in its CMS communications directory that allow the specification of conversation security information. These tags are:

Tag	What Value the Tag Specifies
`:security.`	The security type of the conversation (NONE, SAME, or PGM).
`:userid.`	The access security user ID (used for security type PGM only).
`:password.`	The access security password (used for security type PGM only).

Conversation security information can be specified in other ways as well. The conversation security type can be explicitly set in the program using the Set_Conversation_Security_Type (XCSCST) call. The security user ID and password can be specified in two other ways: using Set calls in the program or by putting an APPCPASS statement in the virtual machine's CP directory. We will discuss these Set calls in the next three sections. See the following FYI box for more information on the APPCPASS statement.

FYI: Security Information and the APPCPASS Statement

If there are concerns about placing security information in a file when SECURITY(PGM) is used, the values can be provided in an APPCPASS statement in the virtual machine's CP directory. An APPCPASS statement entry does not take precedence over a value provided in either side information or on an explicit Set call. Issuing one of the explicit Set calls does override, for the specified conversation, any corresponding information supplied in side information and takes precedence over an APPCPASS statement.

If the security type is SECURITY(PGM) and only a user ID has been provided, or neither a user ID nor a password has been provided, either in side information or through an explicit Set call, then the CP directory is checked for an APPCPASS statement to supply the missing values. When the security type is SECURITY(PGM) and only a password is provided, a CM_PRODUCT_SPECIFIC_ERROR occurs at allocation time.

The following table summarizes when the CP directory is checked for an APPCPASS statement to supply missing values. The first two columns show what, if any, security information is provided in side information or through an explicit Set call. The last column indicates the result.

User ID	Password	Result
yes	no	APPCPASS checked for password, based on user_ID
yes	yes	APPCPASS not checked
no	no	APPCPASS checked for user ID and password, based on LU_name
no	yes	Product-specific error returned on Allocate (CMALLC)

See the *z/VM: Connectivity* book for more information on the APPCPASS statement.

The Set_Conversation_Security_Type (XCSCST) Call

The Set_Conversation_Security_Type (XCSCST) call sets the *conversation_security_type* characteristic for a given conversation, overriding the value assigned with the Initialize_Conversation (CMINIT) call.

Set_Conversation_Security_Type can be called only by the program that initiates a conversation (using the Initialize_Conversation call). The call must be issued while in **Initialize** state, prior to the Allocate (CMALLC) call for the specified conversation.

On VM, a security type can be specified in side information (a communications directory) with the `:security.` tag. If a value is not provided in side information, the default security type of SAME is assigned during conversation initialization. An application needs to issue Set_Conversation_Security_Type only if a value was not set in side information and the default value is not desired.

The format for Set_Conversation_Security_Type is:

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

Input Parameters

Use the ***conversation_ID*** parameter to specify the conversation identifier.

Use the ***conversation_security_type*** parameter to specify the kind of access security information to be sent to the remote LU for validation. If present, the security information consists of either a user ID or a user ID and a password. The *conversation_security_type* can be set to one of these values:

XC_SECURITY_NONE (0)
: indicates that no access security information is being included in the allocation request for the conversation.

XC_SECURITY_SAME (1)
: indicates that the access user ID of the local program's virtual machine is being sent to the target LU and transaction program.

XC_SECURITY_PROGRAM (2)
: indicates that a user ID and password are being supplied and sent in the allocation request for the conversation. The target transaction program can access only the access security user ID.

Output Parameter

Possible values for the ***return_code*** parameter are:

CM_OK (0)
: indicates that the Set_Conversation_Security_Type call executed successfully.

CM_PROGRAM_PARAMETER_CHECK (24)
: indicates that the specified conversation ID is unassigned or that the *conversation_security_type* is set to an undefined value.

CM_PROGRAM_STATE_CHECK (25)
: indicates that the conversation in not in **Initialize** state.

Results of the Call

When *return_code* is anything other than CM_OK, the *conversation_security_type* is unchanged. This call does not cause a state change.

Adding XCSCST to Our Requester Program

Let's set the security type for this conversation to XC_SECURITY_PROGRAM with the Set_Conversation_Security_Type routine. We will add the call immediately following the Initialize_Conversation call in the PROCESS EXEC.

```
/*====================================================================*/
/*  PROCESS EXEC - Sample file requester application.                 */
/*====================================================================*/

   ⋮
/*--------------------------------------------------------------------*/
/* Initialize the conversation.                                       */
/*--------------------------------------------------------------------*/
'CMINIT conversation_ID sym_dest_name return_code'
say; say 'Routine called:  CMINIT'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMINIT'
call TraceParms
/*--------------------------------------------------------------------*/
/* Set the conversation_security_type explicitly.                     */
/*--------------------------------------------------------------------*/
conversation_security_type = XC_SECURITY_PROGRAM
'XCSCST conversation_ID conversation_security_type return_code'
say; say 'Routine called:  XCSCST'
if (return_code ¬= CM_OK) then call ErrorHandler 'XCSCST'
call TraceParms 'conversation_ID conversation_security_type',
```

```
                'return_code'
/*--------------------------------------------------------------------*/
/* Set the partner_LU_name explicitly.                                */
/*--------------------------------------------------------------------*/
partner_LU_name = '*USERID SERVR'
partner_LU_name_length = length(partner_LU_name)
'CMSPLN conversation_ID partner_LU_name',
       'partner_LU_name_length return_code'
say; say 'Routine called:  CMSPLN'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMSPLN'
call TraceParms
   ⋮

/*-------------------------- Subroutines ---------------------------*/

TraceParms:
/*--------------------------------------------------------------------*/
/* Display parameters and their values as passed to this subroutine.*/
/*--------------------------------------------------------------------*/
parse arg parmlist
do word_num = 1 to words(parmlist)
  parameter = word(parmlist,word_num)
  select
    when (parameter = 'return_code') then
      say '  return_code is' cm_return_code.return_code
    when (parameter = 'buffer') then
      say '  buffer is' left(buffer,send_length)
    when (parameter = 'receive_buffer') then
      say '  buffer is' left(receive_buffer,received_length)
    when (parameter = 'data_received') then
      say '  data_received is' cm_data_received.data_received
    when (parameter = 'status_received') then
      say '  status_received is' cm_status_received.status_received
    when (parameter = 'request_to_send_received') then
      say '  request_to_send_received is',
          cm_request_to_send_received.request_to_send_received
    when (parameter = 'sync_level') then
      say '  sync_level is' cm_sync_level.sync_level
    when (parameter = 'prepare_to_receive_type') then
      say '  prepare_to_receive_type is',
          cm_prepare_to_receive_type.prepare_to_receive_type
    when (parameter = 'deallocate_type') then
      say '  deallocate_type is' cm_deallocate_type.deallocate_type
    when (parameter = 'conversation_security_type') then
      say '  conversation_security_type is',
          xc_conversation_security_type.conversation_security_type
    otherwise
      say ' ' parameter 'is' value(parameter)
  end
end
/*--------------------------------------------------------------------*/
/* Extract the current conversation state of the local program.       */
/*--------------------------------------------------------------------*/
/* Commenting out next four lines ...
'CMECS conversation_ID conversation_state return_code'
if (return_code = CM_OK) then
  say '  conversation_state is =>',
      cm_conversation_state.conversation_state
... */

return

   ⋮
```

Recall that we can provide the user ID and password in side information by adding the `:userid.` and the `:password.` tags. Alternatively, we can explicitly provide these

values within the application by using the Set_Conversation_Security_User_ID (XCSCSU) and Set_Conversation_Security_Password (XCSCSP) calls.

The Set_Conversation_Security_User_ID (XCSCSU) Call

The Set_Conversation_Security_User_ID (XCSCSU) call sets the access security user ID for a given conversation.

Set_Conversation_Security_User_ID can be called only by the program that initiates a conversation (using the Initialize_Conversation (CMINIT) call). The call must be issued while in **Initialize** state, prior to the Allocate (CMALLC) call for the specified conversation, and is only valid when the *conversation_security_type* is XC_SECURITY_PROGRAM.

The format for Set_Conversation_Security_User_ID is:

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

Input Parameters

Use the ***conversation_ID*** parameter to specify the conversation identifier.

Use the ***security_user_ID*** parameter to specify the access security user ID that will be sent to the remote LU. This value will be available to the remote transaction program for validation.

Use the ***security_user_ID_length*** parameter to specify the length of the security user ID. This length value can range from zero to eight. If the *security_user_ID_length* is zero, the security user ID is set to null and the *security_user_ID* parameter is ignored.

Output Parameter

Possible values for the ***return_code*** parameter are:

CM_OK (0)
: indicates that the Set_Conversation_Security_User_ID call executed successfully.

CM_PROGRAM_PARAMETER_CHECK (24)
: indicates that the specified conversation ID is unassigned or that the value specified in the *security_user_ID_length* parameter is less than zero or greater than eight.

CM_PROGRAM_STATE_CHECK (25)
: indicates that the conversation is not in **Initialize** state or that the *conversation_security_type* is not XC_SECURITY_PROGRAM.

Results of the Call

When *return_code* indicates CM_OK, the access security user ID specified on this routine overrides a user ID in the communications directory and causes an access security user ID specified in a directory APPCPASS statement to be ignored. If the *security_user_ID_length* parameter is specified as zero, however, the APPCPASS directory statement is checked during allocation processing. This call does not cause a state change.

Adding XCSCSU to Our Requester Program

While an APPCPASS directory statement may be the preferred location for security information in most cases, in our example we will set the access security user ID in the program with the Set_Conversation_Security_User_ID routine.

Let's add the call to the requester's PROCESS EXEC immediately after the Set_Conversation_Security_Type (XCSCST) call. For our example, we will use the requester's user ID, REQUESTR. You will need to substitute the user ID for your requester virtual machine if you used a different name.

```
/*====================================================================*/
/*  PROCESS EXEC - Sample file requester application.                 */
/*====================================================================*/

   ⋮
/*--------------------------------------------------------------------*/
/* Set the conversation_security_type explicitly.                     */
/*--------------------------------------------------------------------*/
conversation_security_type = XC_SECURITY_PROGRAM
'XCSCST conversation_ID conversation_security_type return_code'
say; say 'Routine called:  XCSCST'
if (return_code >= CM_OK) then call ErrorHandler 'XCSCST'
call TraceParms 'conversation_ID conversation_security_type',
                'return_code'
/*--------------------------------------------------------------------*/
/* Set the security_user_ID explicitly.                               */
/*--------------------------------------------------------------------*/
security_user_ID = 'REQUESTR'
security_user_ID_length = length(security_user_ID)
'XCSCSU conversation_ID security_user_ID security_user_ID_length',
       'return_code'
say; say 'Routine called:  XCSCSU'
if (return_code ¬= CM_OK) then call ErrorHandler 'XCSCSU'
call TraceParms 'conversation_ID security_user_ID',
                'security_user_ID_length return_code'
/*--------------------------------------------------------------------*/
/* Set the partner_LU_name explicitly.                                */
/*--------------------------------------------------------------------*/
partner_LU_name = '*USERID SERVR'
partner_LU_name_length = length(partner_LU_name)
'CMSPLN conversation_ID partner_LU_name',
       'partner_LU_name_length return_code'
say; say 'Routine called:  CMSPLN'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMSPLN'
call TraceParms
   ⋮
```

Before we can try the changes we have made, we still need to set the password value.

The Set_Conversation_Security_Password (XCSCSP) Call

The Set_Conversation_Security_Password (XCSCSP) call sets the access security password for a given conversation. The Set_Conversation_Security_Password routine can be called only while in **Initialize** state. This call is valid only when the *conversation_security_type* is XC_SECURITY_PROGRAM.

The format for Set_Conversation_Security_Password is:

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

Input Parameters

Use the ***conversation_ID*** parameter to specify the conversation identifier.

Use the ***security_password*** parameter to specify the access security password that will be passed to the remote LU for validation.

Use the ***security_password_length*** parameter to specify the length of the security password. This length value can range from zero to eight. If the password length is zero, the password is set to null and the *security_password* parameter is ignored.

Output Parameter

Possible values for the ***return_code*** parameter are:

CM_OK (0)
: indicates that the Set_Conversation_Security_Password call executed successfully.

CM_PROGRAM_PARAMETER_CHECK (24)
: indicates that the specified conversation ID is unassigned or that the value specified for the *security_password_length* is less than zero or greater than eight.

CM_PROGRAM_STATE_CHECK (25)
: indicates that the conversation is not in **Initialize** state or the *conversation_security_type* is not XC_SECURITY_PROGRAM.

Results of the Call

When *return_code* indicates CM_OK, the access security password specified on this routine overrides a password in the communications directory and causes an access security password specified in a directory APPCPASS statement to be ignored. If the *security_password_length* parameter is specified as zero, however, the APPCPASS directory statement is checked during allocation processing. This call does not cause a state change.

Adding XCSCSP to Our Requester Program

Having used a Set_Conversation_Security_User_ID (XCSCSU) call to specify the security user ID, we now need to provide the access security password, as well. Let's use the Set_Conversation_Security_Password routine to set that password value in the program.

We will add the call to the requester's exec, following the Set_Conversation_Security_User_ID routine. For the sample program shown in this book we used REQUESTR for the user ID, so we will supply the REQUESTR password. You will need to substitute the password for your requester virtual machine.

```
/*====================================================================*/
/*  PROCESS EXEC - Sample file requester application.                 */
/*====================================================================*/

   ⋮
/*--------------------------------------------------------------------*/
/* Set the security_user_ID explicitly.                               */
/*--------------------------------------------------------------------*/
security_user_ID = 'REQUESTR'
security_user_ID_length = length(security_user_ID)
'XCSCSU conversation_ID security_user_ID security_user_ID_length',
       'return_code'
say; say 'Routine called:  XCSCSU'
if (return_code ¬= CM_OK) then call ErrorHandler 'XCSCSU'
```

```
call TraceParms 'conversation_ID security_user_ID',
                'security_user_ID_length return_code'
/*-------------------------------------------------------------------*/
/* Set the security_password explicitly.                             */
/*-------------------------------------------------------------------*/
security_password = 'PASSWORD'
security_password_length = length(security_password)
'XCSCSP conversation_ID security_password security_password_length',
       'return_code'
say; say 'Routine called:  XCSCSP'
if (return_code ¬= CM_OK) then call ErrorHandler 'XCSCSP'
call TraceParms 'conversation_ID security_password',
                'security_password_length return_code'
/*-------------------------------------------------------------------*/
/* Set the partner_LU_name explicitly.                               */
/*-------------------------------------------------------------------*/
partner_LU_name = '*USERID SERVR'
partner_LU_name_length = length(partner_LU_name)
'CMSPLN conversation_ID partner_LU_name',
       'partner_LU_name_length return_code'
say; say 'Routine called:  CMSPLN'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMSPLN'
call TraceParms
  ⋮
```

We are finally ready to try out our program with the new security level. As always, start things off with

```
process getfile
```

from the requester. The resulting screen display for the requester will be:

```
process getfile
Requesting the file:  TEST FILE A

Routine called:  CMINIT

Routine called:  XCSCST
  conversation_ID is 00000000
  conversation_security_type is XC_SECURITY_PROGRAM
  return_code is CM_OK

Routine called:  XCSCSU
  conversation_ID is 00000000
  security_user_ID is REQUESTR
  security_user_ID_length is 8
  return_code is CM_OK

Routine called:  XCSCSP
  conversation_ID is 00000000
  security_password is PASSWORD
  security_password_length is 8
  return_code is CM_OK

Routine called:  CMSPLN

Routine called:  CMSTPN

Would you like confirmation processing? (Y/N)
N

Routine called:  CMALLC

Routine called:  CMSEND

Routine called:  CMSPTR

Routine called:  CMPTR

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_SEND_RECEIVED

Routine called:  CMDEAL
Ready;
```

Figure 55. Results from PROCESS EXEC

The server will display these lines:

```
Routine called:  XCIDRM

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is
  event_type is XC_ALLOCATION_REQUEST

Routine called:  CMACCP

Routine called:  CMECT

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000000
  event_type is XC_INFORMATION_INPUT

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000000
  event_type is XC_INFORMATION_INPUT

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_SEND_RECEIVED

Routine called:  CMSEND

Routine called:  CMSST

Routine called:  CMSEND

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000000
  event_type is XC_INFORMATION_INPUT

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED
```

Figure 56. Results from SENDBACK EXEC (Part 1 of 2)

```
Conversation deallocated by partner

Waiting for an event to occur.  Enter "QUIT" to exit.
QUIT

Routine called:  XCWOE
  conversation_ID is
  event_type is XC_CONSOLE_INPUT

Routine called:  XCTRRM
Ready;
```

Figure 56. Results from SENDBACK EXEC (Part 2 of 2)

Intermediate Servers

It is time to turn our attention to intermediate servers. An ***intermediate server*** is a program that handles communications requests to a resource manager program on behalf of a user program. The user program can be referred to as a ***client*** of the intermediate server. In our example, the requester PROCESS EXEC allocates a conversation to the server SENDBACK EXEC. Now, if SENDBACK EXEC does not process the file request, but instead allocates a conversation to another program that will actually handle the request, SENDBACK EXEC would take on the role of an intermediate server and PROCESS EXEC would be a client program. This is how the programs we are writing will work, but these are rather simple programs whose purpose is to illustrate how to use the calls.

A more realistic scenario would be a server that, while performing some task for a user, needs to access a resource that is controlled by another resource manager. Security considerations may dictate that the server handling user requests not be authorized to access any other resources on its own. Thus, the server would need to be able to pass along the user ID of the requester for whom it is doing the work so the resource manager controlling the needed resource could determine who was requesting a particular resource.

Another use for an intermediate server in VM might be to control access to a group of resources. This intermediate server could be used to do security validation and to direct user requests to appropriate server virtual machines.

Now we will have three communications programs. The intermediate server needs to be a bit more sophisticated than either of the other two programs. In fact, the requester and the new (final) server will hardly change from the previous section. The following pseudocode summarizes how the intermediate server program will be structured.

Table 5. Overview of Sample Intermediate Server Program

SERVR User ID
Identify_Resource_Manager do forever Wait_on_Event select on event type when allocation request Accept_Conversation (ConvA) if conversation type is basic Send_Error else Initialize_Conversation Allocate (ConvB) when information input -Receive loop- Receive (on conversation in **Receive** state) if confirmation requested Confirmed if complete data received Send_Data (on conversation in **Send** state) if send control received (on conversation in **Receive** state) Prepare_To_Receive (on conversation in **Send** state) -end Receive loop- when console input leave end select end Terminate_Resource_Manager

Setting Up the SERVR2 Virtual Machine

At this point, we are ready to begin using SERVR2, our third user ID. SERVR2 is going to take on the role of the resource manager that has been performed by the SERVR virtual machine. You will need copies of the following SERVR files on SERVR2:

- SENDBACK EXEC (rename to SENDSERV)
- TEST FILE
- PROFILE EXEC

Let's rename the SENDBACK EXEC on SERVR2 to be SENDSERV EXEC. Modify the lines in TEST FILE to indicate that they reside on the final server. The reason we want to copy the PROFILE EXEC over to the new user ID is that it contains some important lines that we added so our server virtual machine could function properly. If you do not want to replace an existing PROFILE EXEC file on the SERVR2 user ID, simply copy the three `SET` commands we added in Chapter 2, "Starter Set CPI Communications Calls."

On SERVR2, create a $SERVER$ NAMES file containing these lines (and substituting your user ID, if appropriate):

```
:nick.GET  :list.SERVR
           :module.SENDSERV
```

Although we are using GET for the nickname in both servers, it is not required that the names be the same. It is just simpler for our purposes in this book.

We will want the resource manager SENDSERV EXEC on the SERVR2 user ID to terminate when the conversation that started it is deallocated. We previously commented out the section of code that issues a LEAVE statement if *return_code* is CM_DEALLOCATED_NORMAL or *status_received* is CM_CONFIRM_DEALLOC_RECEIVED. Let's remove the comments from that part of the program to restore that function.

```
/*====================================================================*/
/*  SENDSERV EXEC - Sample server application.                        */
/*====================================================================*/

   ⋮
/*--------------------------------------------------------------------*/
/* Start continuous Wait_on_Event loop.                               */
/* Any console input will end the loop.                               */
/*--------------------------------------------------------------------*/
requested_file = ''
do forever
  say; say 'Waiting for an event to occur.  Enter "QUIT" to exit.'
  /*------------------------------------------------------------------*/
  /* Issue Wait_on_Event to wait for the next event to occur.         */
  /*------------------------------------------------------------------*/
  'XCWOE resource_ID conversation_ID event_type event_info_length',
        'event_buffer return_code'
  say; say 'Routine called:  XCWOE'
  if (return_code ¬= CM_OK) then call ErrorHandler 'XCWOE'
  call TraceParms 'conversation_ID event_type'
  /*------------------------------------------------------------------*/
  /* Choose next action based on type of event.                       */
  /*------------------------------------------------------------------*/
  select
    when (event_type = XC_ALLOCATION_REQUEST) then
      call AcceptConv
    when (event_type = XC_INFORMATION_INPUT) then
      call ReceiveInfo
    when (event_type = XC_CONSOLE_INPUT) then
      /*--------------------------------------------------------------*/
      /* Leave the Wait_on_Event loop.                                */
      /*--------------------------------------------------------------*/
      leave
    otherwise
      do
        say
        say '* ERROR:  Wait_on_Event reported event_type',
            xc_event_type.event_type
      end
  end /* select */
  /*------------------------------------------------------------------*/
  /* When notice of partner's deallocation is received, leave the     */
  /* Wait_on_Event loop.                                              */
  /*------------------------------------------------------------------*/
  /* Commenting out next three lines ...
  if (return_code = CM_DEALLOCATED_NORMAL) |,
     (status_received = CM_CONFIRM_DEALLOC_RECEIVED) then
    leave
  ... */
end /* do forever */

GetOut:
  call TerminateRes
  exit
```

```
/*------------------------- Subroutines --------------------------*/
  ⋮
```

That completes the set up required for the SENDSERV EXEC to handle file requests on the resource manager virtual machine. Now, let's turn to the SERVR user ID and the program that will become the intermediate server.

Converting the SERVR Virtual Machine into an Intermediate Server

With a little modification, our existing private resource manager program can be turned into an intermediate server program. The intermediate server will be dealing with two distinct conversations. One conversation, ConvA, will be between the requester PROCESS EXEC and the intermediate server SENDBACK EXEC, and the other, ConvB, will be between SENDBACK EXEC and the new resource manager, SERVR2's SENDSERV EXEC, as shown in Figure 57.

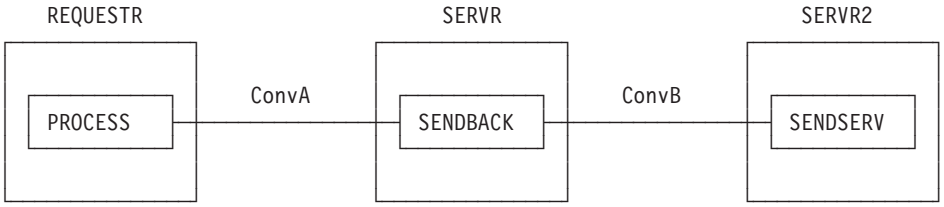


Figure 57. SENDBACK Must Maintain Two Different Conversations

The program, therefore, will need to keep track of two conversation IDs and decide which conversation ID should be used on a particular CPI Communications call.

A summary of the processing to be performed by the SERVR virtual machine follows.

The intermediate server receives an allocation request from the requester program PROCESS EXEC. After accepting the conversation, SENDBACK EXEC can start another conversation with the resource manager program SENDSERV. We will add a new StartConv subroutine to perform that function. In addition, we will include *conversation_ID* on the TraceParms subroutine call (for the Initialize_Conversation (CMINIT) call) in StartConv to see what value gets assigned to the conversation being allocated. Similarly, we will add *conversation_ID* back to the TraceParms call made following the Accept_Conversation call.

Within the ReceiveInfo loop, we will add calls to a new subroutine EndConv that deallocates the conversation with the resource manager when the intermediate server receives notification that the requester program has issued Deallocate (CMDEAL).

Our intermediate server will receive two types of data, the file request from the user program and the file contents from the resource manager. In both cases, the intermediate server needs to forward the data it receives on one conversation to its partner on the other conversation.

When complete data is received, the SendInfo subroutine is called. At that time, the *conversation_ID* variable still identifies the conversation on which data was last received. So, the first step in the SendInfo subroutine is to set *con_ID*, which is the variable that will now be used to specify the conversation ID on the Send_Data call, to the value of the other conversation, as shown in Figure 58 on page 171. In that

way, the program can forward the data to its other partner.

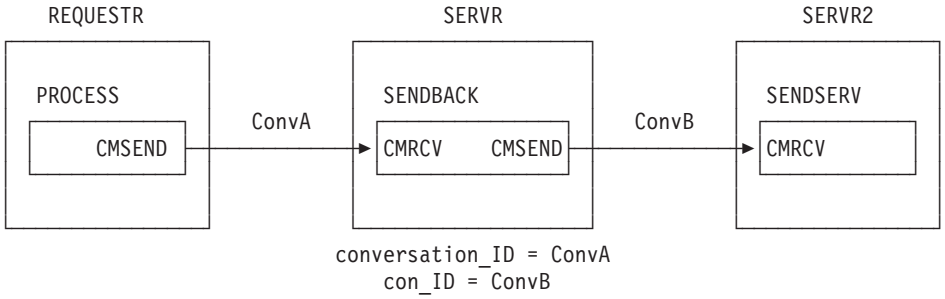


Figure 58. SENDBACK Assigns conversation_ID=ConvA and con_ID=ConvB

As long as one partner is sending data, the intermediate server will need to remain in **Receive** state for the conversation with that partner and remain in **Send** state for the other conversation. This will allow the intermediate server to pass the information it gets on one conversation along to the partner on the other conversation.

We will add a new subroutine named PrepReceive, which will be called when send control notification is received. We will code our programs in such a way that the intermediate server's entering **Send** state on one of its conversations indicates that all the data has been received from the partner that transferred send control. The intermediate server, which forwards data as it receives it, is now finished forwarding data on its other conversation and can transfer send control to the partner on that other conversation. This state change is accomplished with a call to the Prepare_To_Receive subroutine on that other conversation. We will display only the *conversation_ID* on the call to TraceParms.

Because both complete data and send control may be received at the same time, the program needs to protect the *conversation_ID* used on the last Receive call. Hence, the reason for the new variable *con_ID* being used in both SendInfo and PrepReceive subroutines.

In the ReceiveInfo subroutine, when the *data_received* parameter is set to a value other than CM_NO_DATA_RECEIVED, we will want to append the contents of the Receive (CMRCV) buffer to the *send_data* variable to handle partial records. The *send_data* contents will be used later when the data is forwarded to the other partner. We know that for this application, data will be sent to the intermediate server by only one partner at a time, because the other partner is in **Receive** state receiving the forwarded data from the intermediate server. The *send_data* variable is cleared before the Wait_on_Event loop is entered and after a complete data record is forwarded to the other partner.

To make it a little easier to sort out the conversation on which the program is sending data, we will update the TraceParms call for Send_Data to display the *conversation_ID* used.

After a normal termination is received by the intermediate server, the conversation between the file requester and the intermediate server is over. We will want to add a Deallocate call to deallocate the conversation between the intermediate server and the resource manager. Again, we will only show the *conversation_ID* with the call to TraceParms for the new Deallocate call.

Finally, we must update the ErrorHandler subroutine so that it will attempt to abnormally deallocate both conversations if an error is detected.

```
/*====================================================================*/
/*  SENDBACK EXEC - Sample intermediate server application.           */
/*====================================================================*/

  :
/*--------------------------------------------------------------------*/
/* Start continuous Wait_on_Event loop.                               */
/* Any console input will end the loop.                               */
/*--------------------------------------------------------------------*/
send_data = ''
do forever
  say; say 'Waiting for an event to occur.  Enter "QUIT" to exit.'
  /*------------------------------------------------------------------*/
  /* Issue Wait_on_Event to wait for the next event to occur.         */
  /*------------------------------------------------------------------*/
  'XCWOE resource_ID conversation_ID event_type event_info_length',
        'event_buffer return_code'
  say; say 'Routine called:  XCWOE'
  if (return_code ¬= CM_OK) then call ErrorHandler 'XCWOE'
  call TraceParms 'conversation_ID event_type'
  /*------------------------------------------------------------------*/
  /* Choose next action based on type of event.                       */
  /*------------------------------------------------------------------*/
  select
    when (event_type = XC_ALLOCATION_REQUEST) then
      do
        call AcceptConv
        call StartConv
      end
    when (event_type = XC_INFORMATION_INPUT) then
      call ReceiveInfo
    when (event_type = XC_CONSOLE_INPUT) then
      /*----------------------------------------------------------------*/
      /* Leave the Wait_on_Event loop.                                  */
      /*----------------------------------------------------------------*/
      leave
    otherwise
      do
        say
        say '* ERROR:  Wait_on_Event reported event_type',
            xc_event_type.event_type
      end
  end /* select */
  /*------------------------------------------------------------------*/
  /* When notice of partner's deallocation is received, leave the     */
  /* Wait_on_Event loop.                                              */
  /*------------------------------------------------------------------*/
  /* Commenting out next three lines ...
  if (return_code = CM_DEALLOCATED_NORMAL) |,
     (status_received = CM_CONFIRM_DEALLOC_RECEIVED) then
    leave
  ... */
end /* do forever */

GetOut:
  call TerminateRes
  exit

/*------------------------- Subroutines --------------------------*/

AcceptConv:
/*--------------------------------------------------------------------*/
/* Accept the incoming conversation.                                  */
/* Store the conversation identifier in save_con_ID1.                 */
```

```
/*-------------------------------------------------------------------*/
'CMACCP conversation_ID return_code'
save_con_ID1 = conversation_ID
say; say 'Routine called:  CMACCP'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMACCP'
call TraceParms 'conversation_ID'
/*-------------------------------------------------------------------*/
/* Extract conversation_type to ensure the conversation is mapped.  */
/*-------------------------------------------------------------------*/
'CMECT conversation_ID conversation_type return_code'
say; say 'Routine called:  CMECT'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMECT'
call TraceParms
/*-------------------------------------------------------------------*/
/* If the conversation is basic, deallocate abnormally.             */
/*-------------------------------------------------------------------*/
if (conversation_type = CM_BASIC_CONVERSATION) then
  do
    say; say '* ERROR:  Accepting and deallocating a basic',
             'conversation'
    /*---------------------------------------------------------------*/
    /* Call Send_Error to notify partner that error was detected.   */
    /* Since the program is going to exit, do not check the         */
    /* Send_Error results for an error.                             */
    /*---------------------------------------------------------------*/
    'CMSERR conversation_ID request_to_send_received return_code'
    say; say 'Routine called:  CMSERR'
    if (return_code = CM_OK) then
      call TraceParms
    call AbnormalEnd
    signal GetOut
  end

return


StartConv:
/*-------------------------------------------------------------------*/
/* StartConv will establish a conversation with the resource        */
/* manager on behalf of the file requester.                         */
/* First, initialize the conversation to the resource manager.      */
/*-------------------------------------------------------------------*/
sym_dest_name = 'GETFILE'
'CMINIT conversation_ID sym_dest_name return_code'
save_con_ID2 = conversation_ID
say; say 'Routine called:  CMINIT'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMINIT'
call TraceParms 'conversation_ID'
/*-------------------------------------------------------------------*/
/* Allocate conversation. Conversation_ID still equals save_con_ID2.*/
/*-------------------------------------------------------------------*/
'CMALLC conversation_ID return_code'
say; say 'Routine called:  CMALLC'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMALLC'

return


ReceiveInfo:
/*-------------------------------------------------------------------*/
/* Start a Receive loop.                                            */
/* Receive data, status, or both from conversation partner.         */
/*-------------------------------------------------------------------*/
requested_length = event_info_length
do until (data_received = CM_COMPLETE_DATA_RECEIVED) |,
         (data_received = CM_NO_DATA_RECEIVED)
  'CMRCV conversation_ID receive_buffer requested_length',
```

```
        'data_received received_length status_received',
        'request_to_send_received return_code'
    CMRCV_return_code = return_code
    say; say 'Routine called:  CMRCV'
    select
      when (CMRCV_return_code = CM_OK) then
        do
          call TraceParms 'data_received status_received'
          if (data_received ¬= CM_NO_DATA_RECEIVED) then
            do
              receive_buffer = left(receive_buffer,received_length)
              send_data = send_data || receive_buffer
            end
          /*-------------------------------------------------------------*/
          /* Determine whether a confirmation request has been           */
          /* received.  If so, respond with a positive reply.            */
          /*-------------------------------------------------------------*/
          if (status_received = CM_CONFIRM_RECEIVED) |,
             (status_received = CM_CONFIRM_SEND_RECEIVED) |,
             (status_received = CM_CONFIRM_DEALLOC_RECEIVED) then
            do
              /*-------------------------------------------------------*/
              /* Issue Confirmed to reply to the partner.              */
              /*-------------------------------------------------------*/
              'CMCFMD conversation_ID return_code'
              say; say 'Routine called:  CMCFMD'
              if (return_code ¬= CM_OK) then call ErrorHandler 'CMCFMD'
              call TraceParms
            end
          if (data_received = CM_COMPLETE_DATA_RECEIVED) then
            /*-------------------------------------------------------*/
            /* Forward data to partner on the other conversation.    */
            /*-------------------------------------------------------*/
            call SendInfo
          if (status_received = CM_SEND_RECEIVED) |,
             (status_received = CM_CONFIRM_SEND_RECEIVED) then
            /*-------------------------------------------------------*/
            /* The server should only get send control when one      */
            /* partner has completed sending data.                   */
            /*-------------------------------------------------------*/
            call PrepReceive
          else
            if (status_received = CM_CONFIRM_DEALLOC_RECEIVED) then
              do
                say; say 'Conversation deallocated by requester'
                requested_file = ''
                call EndConv
              end
        end
      when (CMRCV_return_code = CM_DEALLOCATED_NORMAL) then
        do
          call TraceParms 'data_received status_received'
          say; say 'Conversation deallocated by requester'
          requested_file = ''
          call EndConv
        end
      otherwise
        call ErrorHandler 'CMRCV'
    end
end

return


SendInfo:
/*-----------------------------------------------------------------*/
/* Send data received on one conversation to partner on other      */
```

```
/* conversation.  The send_data variable contains either the name   */
/* of the requested file or a line from the file, and it was set in */
/* ReceiveInfo.  Conversation_ID was last set on CMRCV call.        */
/* Reset it to the ID of the other conversation.                    */
/*------------------------------------------------------------------*/
if (conversation_ID = save_con_ID1) then
  con_ID = save_con_ID2
else
  con_ID = save_con_ID1
address command 'EXECIO * DISKR' requested_file '(FINIS STEM LINE.'
do index = 1 to line.0
  if (index = line.0) then
    /*----------------------------------------------------------------*/
    /* Reset the send_type conversation characteristic just           */
    /* before the final Send_Data call.                               */
    /*----------------------------------------------------------------*/
    do
      send_type = CM_SEND_AND_PREP_TO_RECEIVE
      'CMSST conversation_ID send_type return_code'
      say; say 'Routine called:  CMSST'
      if (return_code ¬= CM_OK) then call ErrorHandler 'CMSST'
      call TraceParms 'conversation_ID send_type return_code'
    end
buffer = send_data
send_length = length(buffer)
'CMSEND con_ID buffer send_length',
       'request_to_send_received return_code'
say; say 'Routine called:  CMSEND'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMSEND'
call TraceParms 'con_ID'
send_data = ''           /* reset received data variable to nulls */
end

return


PrepReceive:
/*------------------------------------------------------------------*/
/* When send control is received on one conversation, the           */
/* intermediate server is ready to transfer send control to         */
/* partner on the other conversation.                               */
/*------------------------------------------------------------------*/
if (conversation_ID = save_con_ID1) then
  con_ID = save_con_ID2
else
  con_ID = save_con_ID1
'CMPTR con_ID return_code'
say; say 'Routine called:  CMPTR'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMPTR'
call TraceParms 'con_ID'

return


TraceParms:
/*------------------------------------------------------------------*/
/* Display parameters and their values as passed to this subroutine.*/
/*------------------------------------------------------------------*/
parse arg parmlist
do word_num = 1 to words(parmlist)
  parameter = word(parmlist,word_num)
  select
    when (parameter = 'return_code') then
      say '  return_code is' cm_return_code.return_code
    when (parameter = 'buffer') then
      say '  buffer is' left(buffer,send_length)
    when (parameter = 'receive_buffer') then
```

```
      say '  buffer is' left(receive_buffer,received_length)
    when (parameter = 'data_received') then
      say '  data_received is' cm_data_received.data_received
    when (parameter = 'status_received') then
      say '  status_received is' cm_status_received.status_received
    when (parameter = 'request_to_send_received') then
      say '  request_to_send_received is',
          cm_request_to_send_received.request_to_send_received
    when (parameter = 'send_type') then
      say '  send_type is' cm_send_type.send_type
    when (parameter = 'deallocate_type') then
      say '  deallocate_type is' cm_deallocate_type.deallocate_type
    when (parameter = 'conversation_type') then
      say '  conversation_type is',
          cm_conversation_type.conversation_type
    when (parameter = 'resource_manager_type') then
      say '  resource_manager_type is',
          xc_resource_manager_type.resource_manager_type
    when (parameter = 'service_mode') then
      say '  service_mode is' xc_service_mode.service_mode
    when (parameter = 'security_level_flag') then
      say '  security_level_flag is',
          xc_security_level_flag.security_level_flag
    when (parameter = 'event_type') then
      say '  event_type is' xc_event_type.event_type
    otherwise
      say ' ' parameter 'is' value(parameter)
  end
end
/*--------------------------------------------------------------------*/
/* Extract the current conversation state of the local program.       */
/*--------------------------------------------------------------------*/
/* Commenting out next four lines ...
'CMECS conversation_ID conversation_state return_code'
if (return_code = CM_OK) then
  say '  conversation_state is =>',
      cm_conversation_state.conversation_state
... */

return


EndConv:
/*--------------------------------------------------------------------*/
/* Deallocate the conversation with the resource manager.             */
/*--------------------------------------------------------------------*/
conversation_ID = save_con_ID2
'CMDEAL conversation_ID return_code'
say; say 'Routine called:  CMDEAL'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMDEAL'
call TraceParms 'conversation_ID'

return


Error:
/*--------------------------------------------------------------------*/
/* Report error when REXX special variable RC is not 0.               */
/*--------------------------------------------------------------------*/
say
say '* ERROR:  REXX has detected an error'
say '          The return code variable RC was set to' rc
call AbnormalEnd
signal GetOut


ErrorHandler:
```

```
/*-------------------------------------------------------------------*/
/* Report routine that failed and the error return code.             */
/*-------------------------------------------------------------------*/
parse arg routine_name
say
say '* ERROR:  An error occurred during a' routine_name 'call'
say '          The return_code was set to' cm_return_code.return_code
call AbnormalEnd
signal GetOut


AbnormalEnd:
/*-------------------------------------------------------------------*/
/* Abnormally deallocate the conversation.  Since we are exiting     */
/* due to an error, we will not display an error message if the      */
/* Set_Deallocate_Type or Deallocate call encounters an error.       */
/* Use conversation IDs in save_con_ID1 and save_con_ID2.            */
/*-------------------------------------------------------------------*/
deallocate_type = CM_DEALLOCATE_ABEND
conversation_ID = save_con_ID1
'CMSDT conversation_ID deallocate_type return_code'
say; say 'Routine called:  CMSDT'
if (return_code = CM_OK) then
  do
    call TraceParms
    'CMDEAL conversation_ID return_code'
    say; say 'Routine called:  CMDEAL'
    if (return_code = CM_OK) then
      call TraceParms
  end
conversation_ID = save_con_ID2
'CMSDT conversation_ID deallocate_type return_code'
say; say 'Routine called:  CMSDT'
if (return_code = CM_OK) then
  do
    'CMDEAL conversation_ID return_code'
    say; say 'Routine called:  CMDEAL'
  end

return


TerminateRes:
/*-------------------------------------------------------------------*/
/* TerminateRes will terminate ownership of the specified resource. */
/* Use resource name stored in save_res_ID at start of program.      */
/*-------------------------------------------------------------------*/
resource_ID = save_res_ID
'XCTRRM resource_ID return_code'
say; say 'Routine called:  XCTRRM'
if (return_code ¬= CM_OK) then
  do
    say
    say '* ERROR:  An error occurred during an XCTRRM call'
    say '          The return_code was set to',
        cm_return_code.return_code
  end
else
  call TraceParms

return
```

Before we run our programs, we must add location information about the partner program to the intermediate server's side information. Otherwise, when SENDBACK EXEC tries to allocate a conversation to SENDSERV EXEC in the SERVR2 virtual

machine, it will get a CM_TPN_NOT_RECOGNIZED *return_code*. Create a UCOMDIR FILE on SERVR with the following entry (inserting your user ID, if different):

```
:nick.GETFILE  :luname.*USERID SERVR2
               :tpn.GET
```

After filing the UCOMDIR FILE, remember to issue

```
set comdir reload
```

or

```
set comdir file user ucomdir names
```

to put the new information into effect.

To test out the intermediate server, just issue

```
process getfile
```

from the REQUESTR user ID, with no confirmation processing.

The information displayed at the REQUESTR virtual machine will be identical to what we have seen in the past. The terminal display is:

```
process getfile
Requesting the file:  TEST FILE A

Routine called:  CMINIT

Routine called:  XCSCST

Routine called:  XCSCSU

Routine called:  XCSCSP

Routine called:  CMSPLN

Routine called:  CMSTPN

Would you like confirmation processing? (Y/N)
N

Routine called:  CMALLC

Routine called:  CMSEND

Routine called:  CMSPTR

Routine called:  CMPTR

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_SEND_RECEIVED

Routine called:  CMDEAL
Ready;
```

Figure 59. Results from PROCESS EXEC

At the intermediate server virtual machine SERVR, output from both of its conversations will be displayed and should look like:

```
Routine called:  XCIDRM

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is
  event_type is XC_ALLOCATION_REQUEST

Routine called:  CMACCP
  conversation_ID is 00000000

Routine called:  CMECT

Routine called:  CMINIT
  conversation_ID is 00000001

Routine called:  CMALLC

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000000
  event_type is XC_INFORMATION_INPUT

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Routine called:  CMSEND
  con_ID is 00000001

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000000
  event_type is XC_INFORMATION_INPUT

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_SEND_RECEIVED

Routine called:  CMPTR
  con_ID is 00000001

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000001
  event_type is XC_INFORMATION_INPUT

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED
```

Figure 60. Results from Intermediate Server's SENDBACK EXEC (Part 1 of 2)

```
Routine called:  CMSEND
  con_ID is 00000000
Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000001
  event_type is XC_INFORMATION_INPUT

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_SEND_RECEIVED

Routine called:  CMSEND
  con_ID is 00000000

Routine called:  CMPTR
  con_ID is 00000000

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000000
  event_type is XC_INFORMATION_INPUT

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Conversation deallocated by requester

Routine called:  CMDEAL
  conversation_ID is 00000001

Waiting for an event to occur.  Enter "QUIT" to exit.
QUIT

Routine called:  XCWOE
  conversation_ID is
  event_type is XC_CONSOLE_INPUT

Routine called:  XCTRRM
Ready;
```

Figure 60. Results from Intermediate Server's SENDBACK EXEC (Part 2 of 2)

The SERVR2 display will be essentially identical to the results we have seen for the resource manager in the past. The results at the SERVR2 user ID are:

```
Routine called:  XCIDRM

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is
  event_type is XC_ALLOCATION_REQUEST

Routine called:  CMACCP

Routine called:  CMECT

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000000
  event_type is XC_INFORMATION_INPUT

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000000
  event_type is XC_INFORMATION_INPUT

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_SEND_RECEIVED

Routine called:  CMSEND

Routine called:  CMSST

Routine called:  CMSEND

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000000
  event_type is XC_INFORMATION_INPUT

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Conversation deallocated by partner

Routine called:  XCTRRM
Ready;
```

Figure 61. Results from SERV2's SENDSERV EXEC

Security Considerations for Intermediate Servers

Now that we have modified a server program to act as an intermediate server, let's turn our attention back to the topic of security. Before we created the intermediate server, the requester program was communicating directly with the resource manager. If the *conversation_security_type* was either XC_SECURITY_PROGRAM or the default XC_SECURITY_SAME, the resource manager application was sent an access security user ID.

An application can use this access security user ID to determine whether the requester program is authorized to use a particular resource. For example, the $SERVER$ NAMES file entry for a private resource manager may include an asterisk on the `:list.` tag, meaning that any application is authorized to connect to the resource manager. The resource manager, however, may control some resources, such as restricted files, that only certain users are allowed to access. When a request is received for one of those resources, the resource manager can use the Extract_Conversation_Security_User_ID (XCECSU) call to determine if the requester is authorized to access it.

What happens to the access security user ID when an intermediate server is used? If the intermediate server is considered to be a TP-model application, then the user ID of the virtual machine running the requester program will be sent to the resource manager, as shown in Figure 62.

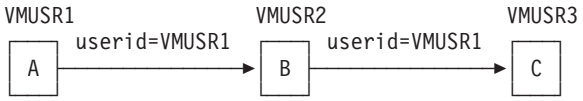


Figure 62. Requester's User ID Is Sent to VMUSR3 with TP-Model Application B's Allocate

If the intermediate server is not a TP-model application (like in our example program, because we call Identify_Resource_Manager), the access security user ID sent to the resource manager is the user ID of the intermediate server, as shown in Figure 63.

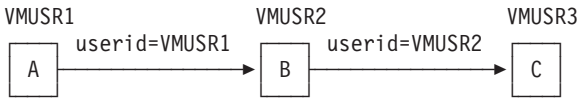


Figure 63. Access Security User ID of Intermediate Server (VMUSR2) Sent to VMUSR3

If the server controlling the resource needs the access security user ID, the non-TP-model intermediate server can issue a Set_Client_Security_User_ID (XCSCUI) call so that the user ID of the requester program's virtual machine is sent to the resource manager. That call requires that the intermediate server's virtual machine have authorization to issue DIAGNOSE code X'D4'. This is typically Class B privilege, unless the default privilege classes have been changed.

We will look at Set_Client_Security_User_ID in a moment. First, though, let's discuss the call that can provide an intermediate server application with the requester program's access security user ID.

The Extract_Conversation_Security_User_ID (XCECSU) Call

The Extract_Conversation_Security_User_ID (XCECSU) call is used by a program to extract the access security user ID associated with a given conversation.

A security user ID will be returned only when the *conversation_security_type* for the conversation is set to XC_SECURITY_SAME or XC_SECURITY_PROGRAM. The *security_user_ID* parameter will contain nulls (X'00') if Extract_Conversation_Security_User_ID is issued when the *conversation_security_type* is XC_SECURITY_NONE.

A call to this routine does not change the conversation security user ID for the specified conversation.

The format for Extract_Conversation_Security_User_ID is:

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

Input Parameter

Use the ***conversation_ID*** parameter to identify the conversation.

Output Parameters

The ***security_user_ID*** parameter is a variable for returning the access security user ID of the conversation partner. If the *return_code* is not set to CM_OK, the *security_user_ID* will not contain a meaningful value.

The ***security_user_ID_length*** parameter returns the length of the security user ID.

Possible values for the ***return_code*** parameter are:

CM_OK (0)
: indicates that the Extract_Conversation_Security_User_ID (XCECSU) call executed successfully.

CM_PROGRAM_PARAMETER_CHECK (24)
: indicates that the specified conversation ID is unassigned.

Results of the Call

When *return_code* indicates CM_OK the access security user ID associated with the conversation is returned. This call does not cause a state change.

Adding XCECSU to Our Intermediate Server Program

Let's add the Extract_Conversation_Security_User_ID call to the intermediate server as the first step in sending the user program's user ID (REQUESTR) to the resource manager (SERVR2). We will include the new call at the end of the AcceptConv subroutine. Only the *security_user_ID* will be passed to TraceParms.

The changes to the SENDBACK EXEC on the SERVR user ID are:

```
/*================================================================*/
/*  SENDBACK EXEC - Sample intermediate server application.       */
/*================================================================*/

  ⋮

/*------------------------- Subroutines --------------------------*/
```

```
AcceptConv:
/*-------------------------------------------------------------------*/
/* Accept the incoming conversation.                                 */
/* Store the conversation identifier in save_con_ID1.                */
/*-------------------------------------------------------------------*/
'CMACCP conversation_ID return_code'
save_con_ID1 = conversation_ID
say; say 'Routine called:  CMACCP'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMACCP'
call TraceParms 'conversation_ID'
/*-------------------------------------------------------------------*/
/* Extract conversation_type to ensure the conversation is mapped.  */
/*-------------------------------------------------------------------*/
'CMECT conversation_ID conversation_type return_code'
say; say 'Routine called:  CMECT'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMECT'
call TraceParms
/*-------------------------------------------------------------------*/
/* If the conversation is basic, deallocate abnormally.              */
/*-------------------------------------------------------------------*/
if (conversation_type = CM_BASIC_CONVERSATION) then
  do
    say; say '* ERROR:  Accepting and deallocating a basic',
             'conversation'
    /*---------------------------------------------------------------*/
    /* Call Send_Error to notify partner that error was detected.   */
    /* Since the program is going to exit, do not check the         */
    /* Send_Error results for an error.                              */
    /*---------------------------------------------------------------*/
    'CMSERR conversation_ID request_to_send_received return_code'
    say; say 'Routine called:  CMSERR'
    if (return_code = CM_OK) then
      call TraceParms
    call AbnormalEnd
    signal GetOut
  end
/*-------------------------------------------------------------------*/
/* Extract the access security user ID for the conversation.        */
/*-------------------------------------------------------------------*/
'XCECSU conversation_ID security_user_ID security_user_ID_length',
      'return_code'
say; say 'Routine called:  XCECSU'
if (return_code ¬= CM_OK) then call ErrorHandler 'XCECSU'
call TraceParms 'security_user_ID'

return


StartConv:
/*-------------------------------------------------------------------*/
/* StartConv will establish a conversation with the resource        */
/* manager on behalf of the file requester.                          */
/* First, initialize the conversation to the resource manager.      */
/*-------------------------------------------------------------------*/
sym_dest_name = 'GETFILE'
'CMINIT conversation_ID sym_dest_name return_code'
save_con_ID2 = conversation_ID
say; say 'Routine called:  CMINIT'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMINIT'
call TraceParms 'conversation_ID'
/*-------------------------------------------------------------------*/
/* Allocate conversation. Conversation_ID still equals save_con_ID2.*/
/*-------------------------------------------------------------------*/
'CMALLC conversation_ID return_code'
say; say 'Routine called:  CMALLC'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMALLC'
call TraceParms
```

```
return

  ⋮
```

Adding XCECSU to Our Resource Manager Program

By adding the Extract_Conversation_Security_User_ID call to the resource manager as well, we can observe what access security user ID is being sent to it. The application could use the value returned from that call to determine if the conversation should be continued for that particular user ID.

Let's insert the call in the same location, at the end of the AcceptConv subroutine. The SENDSERV EXEC update for SERVR2 is:

```
/*====================================================================*/
/*  SENDSERV EXEC - Sample server application.                        */
/*====================================================================*/

  ⋮

/*-------------------------- Subroutines --------------------------*/

AcceptConv:
/*--------------------------------------------------------------------*/
/* Accept the incoming conversation.                                  */
/* Store the conversation identifier in save_con_ID.                  */
/*--------------------------------------------------------------------*/
'CMACCP conversation_ID return_code'
save_con_ID = conversation_ID
say; say 'Routine called:  CMACCP'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMACCP'
call TraceParms
/*--------------------------------------------------------------------*/
/* Extract conversation_type to ensure the conversation is mapped.  */
/*--------------------------------------------------------------------*/
'CMECT conversation_ID conversation_type return_code'
say; say 'Routine called:  CMECT'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMECT'
call TraceParms
/*--------------------------------------------------------------------*/
/* If the conversation is basic, deallocate abnormally.               */
/*--------------------------------------------------------------------*/
if (conversation_type = CM_BASIC_CONVERSATION) then
  do
    say; say '* ERROR:  Accepting and deallocating a basic',
             'conversation'
    /*----------------------------------------------------------------*/
    /* Call Send_Error to notify partner that error was detected.   */
    /* Since the program is going to exit, do not check the         */
    /* Send_Error results for an error.                             */
    /*----------------------------------------------------------------*/
    'CMSERR conversation_ID request_to_send_received return_code'
    say; say 'Routine called:  CMSERR'
    if (return_code = CM_OK) then
      call TraceParms
    call AbnormalEnd
    signal GetOut
  end
/*--------------------------------------------------------------------*/
/* Extract the access security user ID for the conversation.        */
/*--------------------------------------------------------------------*/
'XCECSU conversation_ID security_user_ID security_user_ID_length',
       'return_code'
say; say 'Routine called:  XCECSU'
if (return_code ¬= CM_OK) then call ErrorHandler 'XCECSU'
call TraceParms 'security_user_ID'
```

```
return


ReceiveInfo:
/*-------------------------------------------------------------------*/
/* Start a Receive loop.                                             */
/* Receive data, status, or both from conversation partner.          */
/*-------------------------------------------------------------------*/
requested_length = event_info_length
   :
```

Again, we will try out the results by entering

```
process getfile
```

from the REQUESTR command line, and no confirmation processing will be requested.

The PROCESS EXEC output on the REQUESTR user ID is unchanged:

```
process getfile
Requesting the file:  TEST FILE A

Routine called:  CMINIT

Routine called:  XCSCST

Routine called:  XCSCSU

Routine called:  XCSCSP

Routine called:  CMSPLN

Routine called:  CMSTPN

Would you like confirmation processing? (Y/N)
N

Routine called:  CMALLC

Routine called:  CMSEND

Routine called:  CMSPTR

Routine called:  CMPTR

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_SEND_RECEIVED

Routine called:  CMDEAL
Ready;
```

Figure 64. Results from Requester's PROCESS EXEC

On the SERVR virtual machine, the intermediate server results are:

```
Routine called:  XCIDRM

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is
  event_type is XC_ALLOCATION_REQUEST

Routine called:  CMACCP
  conversation_ID is 00000000

Routine called:  CMECT

Routine called:  XCECSU
  security_user_ID is REQUESTR

Routine called:  CMINIT
  conversation_ID is 00000001

Routine called:  CMALLC

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000000
  event_type is XC_INFORMATION_INPUT

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Routine called:  CMSEND
  con_ID is 00000001

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000000
  event_type is XC_INFORMATION_INPUT

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_SEND_RECEIVED

Routine called:  CMPTR
  con_ID is 00000001

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000001
  event_type is XC_INFORMATION_INPUT
```

Figure 65. Results from Intermediate Server's SENDBACK EXEC (Part 1 of 2)

```
Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Routine called:  CMSEND
  con_ID is 00000000

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000001
  event_type is XC_INFORMATION_INPUT

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_SEND_RECEIVED

Routine called:  CMSEND
  con_ID is 00000000

Routine called:  CMPTR
  con_ID is 00000000

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000000
  event_type is XC_INFORMATION_INPUT

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Conversation deallocated by requester

Routine called:  CMDEAL
  conversation_ID is 00000001

Waiting for an event to occur.  Enter "QUIT" to exit.
QUIT

Routine called:  XCWOE
  conversation_ID is
  event_type is XC_CONSOLE_INPUT

Routine called:  XCTRRM
Ready;
```

Figure 65. Results from Intermediate Server's SENDBACK EXEC (Part 2 of 2)

Lastly, displayed at the SERVR2 user ID is:

```
Routine called:  XCIDRM

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is
  event_type is XC_ALLOCATION_REQUEST

Routine called:  CMACCP

Routine called:  CMECT

Routine called:  XCECSU
  security_user_ID is SERVR

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000000
  event_type is XC_INFORMATION_INPUT

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000000
  event_type is XC_INFORMATION_INPUT

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_SEND_RECEIVED

Routine called:  CMSEND

Routine called:  CMSST

Routine called:  CMSEND

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000000
  event_type is XC_INFORMATION_INPUT

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Conversation deallocated by partner

Routine called:  XCTRRM
Ready;
```

Figure 66. Results from SERVR2's SENDSERV EXEC

Notice that the access security user ID extracted by the SERVR's SENDBACK EXEC is REQUESTR, and the value extracted by the SERVR2's SENDSERV EXEC is SERVR. Thus, SENDSERV has no way of knowing that the request originated from the REQUESTR virtual machine.

Because the intermediate server is simply passing a request for data along to the resource manager, it may be desirable to let the resource manager know the value of the access security user ID used for the initial conversation from the file requester to the intermediate server. The next routine provides a way to forward an access security user ID, provided the virtual machine has the required privilege class.

The Set_Client_Security_User_ID (XCSCUI) Call

The Set_Client_Security_User_ID (XCSCUI) call is used by an intermediate server to set an access security user ID value for a given conversation based on an incoming conversation's access security user ID. This user ID is then presented to the target when the intermediate server allocates a conversation on behalf of the client application (the requester program).

An intermediate server may have incoming conversations from various virtual machines. Set_Client_Security_User_ID (XCSCUI) can be used to specify a particular user ID that will be presented to the target resource manager. In this way, the target resource manager virtual machine knows the origin of the request.

An intermediate server can call Set_Client_Security_User_ID (XCSCUI) only when the following conditions are true:

- The program is not a TP-model application.
- The specified conversation (to be allocated to the target resource manager) has a *conversation_security_type* characteristic equal to XC_SECURITY_SAME.
- An access security user ID is available for the incoming conversation with the client. The access security user ID for that conversation should be retrieved by calling the Extract_Conversation_Security_User_ID (XCECSU) routine.
- The intermediate server virtual machine is authorized to issue a DIAGNOSE code X'D4' (for defining an alternate user ID). This authorization typically requires privilege class B, unless the default privilege classes have been altered. If not authorized, the Allocate (CMALLC) call will complete with a CM_PRODUCT_SPECIFIC_ERROR return code.

This call can be issued only when the specified conversation is in **Initialize** state, prior to the Allocate call.

The format for Set_Client_Security_User_ID is:

```
CALL  XCSCUI(conversation_ID,          input
             client_user_ID,           input
             return_code)              output
```

Input Parameters

There are two input parameters. Specify the identifier for the outgoing conversation in the ***conversation_ID*** parameter.

Use the ***client_user_ID*** parameter to specify the user ID that was obtained using Extract_Conversation_Security_User_ID.

Output Parameter

Possible values for the ***return_code*** parameter are:

CM_OK (0)
: indicates that Set_Client_Security_User_ID executed successfully.

CM_PROGRAM_PARAMETER_CHECK (24)
: indicates that the specified conversation ID is unassigned or that the *conversation_security_type* of the outgoing conversation is not XC_SECURITY_SAME.

CM_PROGRAM_STATE_CHECK (25)
: indicates that the conversation in not in **Initialize** state or that the program is a TP-model application, which makes this call invalid.

Results of the Call

Upon successful completion, the *security_user_ID* for the specified conversation is set to the value specified on the call. This call does not cause a state change.

If the intermediate server virtual machine is not authorized to issue DIAGNOSE code X'D4', an allocation error results.

Adding XCSCUI to Our Intermediate Server Program

Providing your intermediate server virtual machine is authorized to issue DIAGNOSE code X'D4', you can add the Set_Client_Security_User_ID call to the intermediate server application. By adding that call, we can have the access security user ID associated with the requester's conversation sent to the resource manager. Only the *client_user_ID* will be passed for TraceParms to display.

We will add the call in the StartConv subroutine, just after the Initialize_Conversation (CMINIT) call. The SERVR file SENDBACK EXEC is updated, as follows:

```
/*====================================================================*/
/*  SENDBACK EXEC - Sample intermediate server application.           */
/*====================================================================*/

  ⋮

StartConv:
/*--------------------------------------------------------------------*/
/* StartConv will establish a conversation with the resource          */
/* manager on behalf of the file requester.                           */
/* First, initialize the conversation to the resource manager.        */
/*--------------------------------------------------------------------*/
sym_dest_name = 'GETFILE'
'CMINIT conversation_ID sym_dest_name return_code'
save_con_ID2 = conversation_ID
say; say 'Routine called:  CMINIT'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMINIT'
call TraceParms 'conversation_ID'
/*--------------------------------------------------------------------*/
/* Pass the requesting program's access user ID to resource manager.*/
/*--------------------------------------------------------------------*/
client_user_ID = security_user_ID
'XCSCUI conversation_ID client_user_ID return_code'
say; say 'Routine called:  XCSCUI'
if (return_code ¬= CM_OK) then call ErrorHandler 'XCSCUI'
call TraceParms 'client_user_ID'
/*--------------------------------------------------------------------*/
/* Allocate conversation. Conversation_ID still equals save_con_ID2.*/
/*--------------------------------------------------------------------*/
'CMALLC conversation_ID return_code'
say; say 'Routine called:  CMALLC'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMALLC'
call TraceParms

return
```

⋮

After entering

```
process getfile
```

at the REQUESTR user ID, we will rather quickly encounter a problem. At the resource manager's virtual machine, SERVR2, a familiar error message appears (with the corresponding time):

```
hh:mm:ss  * MSG FROM SERVR2  : DMSIUH2027E Connection request on path 0
is severed for reason = 7
```

Figure 67. Results at SERVR2's Console

FYI: If SERVR Received a Product-Specific Error Instead

If the following message was appended to the CPICOMM LOGDATA file on the SERVR user ID (intermediate server):

```
CMALLC_PRODUCT_SPECIFIC_ERROR: Unable to set alternate user ID
```

the VM system you are working on may have an external security manager program such as Resource Access Control Facility/Virtual Machine (RACF*/VM) installed. Such a program may require a special command to be entered before it will allow an alternate user ID to be set. You will need to check the documentation for that external security manager to determine how to obtain the necessary authority for the intermediate server.

This sever message indicates that some validation attempt has failed and suggests that there may be a missing entry in the $SERVER$ NAMES file. Apparently, the Set_Client_Security_User_ID call really did make it appear to the resource manager that the file request was coming directly from the REQUESTR user ID. However, only the intermediate server, SERVR, is authorized in the $SERVER$ NAMES file to connect to the resource manager SENDSERV EXEC, so the conversation was deallocated.

A quick update to the $SERVER$ NAMES file on the SERVR2 virtual machine will clear up this problem. Let's simply add REQUESTR to the list of authorized users of the resource manager:

```
:nick.GET  :list.SERVR REQUESTR
           :module.SENDSERV
```

Starting our programs again will yield these results at the REQUESTR terminal:

```
process getfile
Requesting the file:  TEST FILE A

Routine called:  CMINIT

Routine called:  XCSCST

Routine called:  XCSCSU

Routine called:  XCSCSP

Routine called:  CMSPLN

Routine called:  CMSTPN

Would you like confirmation processing? (Y/N)
N

Routine called:  CMALLC

Routine called:  CMSEND

Routine called:  CMSPTR

Routine called:  CMPTR

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_SEND_RECEIVED

Routine called:  CMDEAL
Ready;
```

Figure 68. Results from Requester's PROCESS EXEC

The SENDBACK EXEC on the SERVR virtual machine displays the following output:

```
Routine called:  XCIDRM

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is
  event_type is XC_ALLOCATION_REQUEST

Routine called:  CMACCP
  conversation_ID is 00000000

Routine called:  CMECT

Routine called:  XCECSU
  security_user_ID is REQUESTR

Routine called:  CMINIT
  conversation_ID is 00000001

Routine called:  XCSCUI
  client_user_ID is REQUESTR

Routine called:  CMALLC

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000000
  event_type is XC_INFORMATION_INPUT

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Routine called:  CMSEND
  con_ID is 00000001

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000000
  event_type is XC_INFORMATION_INPUT

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_SEND_RECEIVED

Routine called:  CMPTR
  con_ID is 00000001
```

Figure 69. Results from Server's SENDBACK EXEC (Part 1 of 2)

```
Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000001
  event_type is XC_INFORMATION_INPUT

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Routine called:  CMSEND
  con_ID is 00000000

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000001
  event_type is XC_INFORMATION_INPUT

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_SEND_RECEIVED

Routine called:  CMSEND
  con_ID is 00000000

Routine called:  CMPTR
  con_ID is 00000000

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000000
  event_type is XC_INFORMATION_INPUT

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Conversation deallocated by requester

Routine called:  CMDEAL
  conversation_ID is 00000001

Waiting for an event to occur.  Enter "QUIT" to exit.
QUIT

Routine called:  XCWOE
  conversation_ID is
  event_type is XC_CONSOLE_INPUT

Routine called:  XCTRRM
Ready;
```

Figure 69. Results from Server's SENDBACK EXEC (Part 2 of 2)

And, final results from the resource manager virtual machine, SERVR2, are:

```
Routine called:  XCIDRM

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is
  event_type is XC_ALLOCATION_REQUEST

Routine called:  CMACCP

Routine called:  CMECT

Routine called:  XCECSU
  security_user_ID is REQUESTR

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000000
  event_type is XC_INFORMATION_INPUT

Routine called:  CMRCV
  data_received is CM_COMPLETE_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000000
  event_type is XC_INFORMATION_INPUT

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_SEND_RECEIVED

Routine called:  CMSEND

Routine called:  CMSST

Routine called:  CMSEND

Waiting for an event to occur.  Enter "QUIT" to exit.

Routine called:  XCWOE
  conversation_ID is 00000000
  event_type is XC_INFORMATION_INPUT

Routine called:  CMRCV
  data_received is CM_NO_DATA_RECEIVED
  status_received is CM_NO_STATUS_RECEIVED

Conversation deallocated by partner

Routine called:  XCTRRM
Ready;
```

Figure 70. Results from SERVR2's SENDSERV EXEC

Now the LU where SENDSERV resides is able to validate the original requester against the entries in the SERVR2 user ID's $SERVER$ NAMES file to ensure that only authorized users are allowed to access the resources controlled by SENDSERV.

Overview of Additional VM Extension Calls

None of the routines covered in this section are used in any of our sample programs. They are included here to give you a brief introduction to the other extension routines available to programmers in the VM environment.

Extract_Conversation_LUWID (XCECL) Call

The Extract_Conversation_LUWID (XCECL) call extracts the logical unit of work ID (LUWID) associated with the specified protected conversation. The LUWID can be used to identify the most recent synchronization point. This routine can be called only after an Allocate (CMALLC) or Accept_Conversation (CMACCP) call that establishes a protected (*sync_level* value of CM_SYNC_POINT) conversation.

Extract_Conversation_Workunitid (XCECWU) Call

The Extract_Conversation_Workunitid (XCECWU) call extracts the CMS work unit ID associated with the specified conversation. Extract_Conversation_Workunitid is especially useful for resource managers that handle multiple requests for multiple resources.

The output from this routine can be used as input to specify the work unit ID on such CSL routines as Push Workunitid (DMSPUSH), for changing the default CMS work unit, and Commit (DMSCOMM) and Rollback (DMSROLLB) when using Coordinated Resource Recovery. These callable services library (CSL) routines are described in the *z/VM: CMS Callable Services Reference*. For information on CMS work units, refer to the *z/VM: CMS Application Development Guide*.

Extract_Local_Fully_Qualified_LU_Name (XCELFQ) Call

The Extract_Local_Fully_Qualified_LU_Name (XCELFQ) call extracts the local fully-qualified LU name for the specified conversation. The output from this routine can be used as input on the Resource Adapter Registration (DMSREG) CSL routine, which is described in the *z/VM: CMS Callable Services Reference*.

Extract_Remote_Fully_Qualified_LU_Name (XCERFQ) Call

The Extract_Remote_Fully_Qualified_LU_Name (XCERFQ) call extracts the remote fully-qualified LU name for the specified conversation. The output from this routine can be used as input on the Resource Adapter Registration (DMSREG) CSL routine, which is described in the *z/VM: CMS Callable Services Reference*.

Extract_TP_Name (XCETPN) Call

The Extract_TP_Name (XCETPN) call extracts the TP name characteristic for the specified conversation.

Signal_User_Event (XCSUE) Call

The Signal_User_Event (XCSUE) call queues an event to be reported by a subsequent Wait_on_Event call in the same virtual machine. Signal_User_Event would typically be called from an event handler to let a CPI Communications program running in the same virtual machine know about some event such as the receipt of a message or the lapsing of a time interval. The CPI Communications program would have to issue Wait_on_Event (XCWOE) to get the user-event notification.

The Completed Sample Execs

Listings of the three completed programs are provided in this section. All of the changes made during this chapter are incorporated into these final versions.

The PROCESS Sample File Requester Exec

```
/*====================================================================*/
/*  PROCESS EXEC - Sample file requester application.                 */
/*====================================================================*/

arg sym_dest_name fname ftype fmode .           /* get user's input */
/*--------------------------------------------------------------------*/
/* If a file was not specifically requested, set up a default.        */
/*--------------------------------------------------------------------*/
if (fname = '') then
  do
    fname = 'TEST'
    ftype = 'FILE'
    fmode = 'A'
  end
say 'Requesting the file: ' fname ftype fmode
/*--------------------------------------------------------------------*/
/* Set up REXX environment for program-to-program communications      */
/* and enable error trapping of REXX errors.                          */
/*--------------------------------------------------------------------*/
address cpicomm
signal on error
/*--------------------------------------------------------------------*/
/* Equate pseudonyms to their integer values based on the             */
/* definitions contained in the CMREXX COPY file.                     */
/*--------------------------------------------------------------------*/
address command 'EXECIO * DISKR CMREXX COPY * (FINIS STEM PSEUDONYM.'
do index = 1 to pseudonym.0
  interpret pseudonym.index
end
/*--------------------------------------------------------------------*/
/* Initialize the conversation.                                       */
/*--------------------------------------------------------------------*/
'CMINIT conversation_ID sym_dest_name return_code'
say; say 'Routine called:  CMINIT'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMINIT'
call TraceParms
/*--------------------------------------------------------------------*/
/* Set the conversation_security_type explicitly.                     */
/*--------------------------------------------------------------------*/
conversation_security_type = XC_SECURITY_PROGRAM
'XCSCST conversation_ID conversation_security_type return_code'
say; say 'Routine called:  XCSCST'
if (return_code ¬= CM_OK) then call ErrorHandler 'XCSCST'
call TraceParms
/*--------------------------------------------------------------------*/
/* Set the security_user_ID explicitly.                               */
/*--------------------------------------------------------------------*/
security_user_ID = 'REQUESTR'
security_user_ID_length = length(security_user_ID)
'XCSCSU conversation_ID security_user_ID security_user_ID_length',
       'return_code'
say; say 'Routine called:  XCSCSU'
if (return_code ¬= CM_OK) then call ErrorHandler 'XCSCSU'
call TraceParms
/*--------------------------------------------------------------------*/
/* Set the security_password explicitly.                              */
/*--------------------------------------------------------------------*/
security_password = 'PASSWORD'
security_password_length = length(security_password)
```

```
'XCSCSP conversation_ID security_password security_password_length',
       'return_code'
say; say 'Routine called:  XCSCSP'
if (return_code ¬= CM_OK) then call ErrorHandler 'XCSCSP'
call TraceParms
/*-------------------------------------------------------------------*/
/* Set the partner_LU_name explicitly.                               */
/*-------------------------------------------------------------------*/
partner_LU_name = '*USERID SERVR'
partner_LU_name_length = length(partner_LU_name)
'CMSPLN conversation_ID partner_LU_name',
       'partner_LU_name_length return_code'
say; say 'Routine called:  CMSPLN'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMSPLN'
call TraceParms
/*-------------------------------------------------------------------*/
/* Set the transaction program name (TP_name) explicitly.            */
/*-------------------------------------------------------------------*/
TP_name = 'GET'
TP_name_length = length(TP_name)
'CMSTPN conversation_ID TP_name TP_name_length return_code'
say; say 'Routine called:  CMSTPN'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMSTPN'
call TraceParms
/*-------------------------------------------------------------------*/
/* Determine if confirmation processing is desired.                  */
/*-------------------------------------------------------------------*/
say; say 'Would you like confirmation processing? (Y/N)'
parse upper pull perform_confirm
if (perform_confirm = 'Y') then
  do
    /*---------------------------------------------------------------*/
    /* Set sync_level to CM_CONFIRM.                                 */
    /*---------------------------------------------------------------*/
    sync_level = CM_CONFIRM
    'CMSSL conversation_ID sync_level return_code'
    say; say 'Routine called:  CMSSL'
    if (return_code ¬= CM_OK) then call ErrorHandler 'CMSSL'
    call TraceParms
    say '  Confirmation processing enabled'
  end
/*-------------------------------------------------------------------*/
/* Allocate the conversation.                                        */
/*-------------------------------------------------------------------*/
'CMALLC conversation_ID return_code'
say; say 'Routine called:  CMALLC'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMALLC'
call TraceParms
/*-------------------------------------------------------------------*/
/* Send the name of the file being requested to the partner program.*/
/*-------------------------------------------------------------------*/
buffer = fname ftype fmode
send_length = length(buffer)
'CMSEND conversation_ID buffer send_length',
       'request_to_send_received return_code'
say; say 'Routine called:  CMSEND'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMSEND'
call TraceParms
/*-------------------------------------------------------------------*/
/* Call Confirm only when sync_level is not CM_NONE.  We can use     */
/* the confirmation processing flag set from console input.          */
/*-------------------------------------------------------------------*/
if (perform_confirm = 'Y') then
  do
    /*---------------------------------------------------------------*/
    /* Confirm that partner has started and received the name of     */
    /* the requested file.                                           */
```

```
    /*-----------------------------------------------------------*/
    'CMCFM conversation_ID request_to_send_received',
          'return_code'
    say; say 'Routine called:  CMCFM'
    if (return_code ¬= CM_OK) then call ErrorHandler 'CMCFM'
    call TraceParms
  end
/*-------------------------------------------------------------------*/
/* Set the prepare_to_receive_type to CM_PREP_TO_RECEIVE_FLUSH.      */
/*-------------------------------------------------------------------*/
prepare_to_receive_type = CM_PREP_TO_RECEIVE_FLUSH
'CMSPTR conversation_ID prepare_to_receive_type return_code'
say; say 'Routine called:  CMSPTR'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMSPTR'
call TraceParms
/*-------------------------------------------------------------------*/
/* Issue Prepare_To_Receive to switch the conversation state from    */
/* Send state to Receive state.                                      */
/*-------------------------------------------------------------------*/
'CMPTR conversation_ID return_code'
say; say 'Routine called:  CMPTR'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMPTR'
call TraceParms
/*-------------------------------------------------------------------*/
/* Start a Receive loop.  Receive calls will be issued until         */
/* notification that the partner has finished sending data and       */
/* entered Receive state at its end of the conversation (noted by    */
/* receipt of CM_SEND_RECEIVED or CM_CONFIRM_SEND_RECEIVED           */
/* for status_received) or until a return_code value other than      */
/* CM_OK is returned.  The record length of the incoming data        */
/* is assumed to be 80 bytes, or less.                               */
/*-------------------------------------------------------------------*/
complete_line = ''
requested_length = 80
do until (status_received = CM_SEND_RECEIVED) |,
         (status_received = CM_CONFIRM_SEND_RECEIVED)
  /*-----------------------------------------------------------------*/
  /* Receive information from the conversation partner.              */
  /*-----------------------------------------------------------------*/
  'CMRCV conversation_ID receive_buffer requested_length',
        'data_received received_length status_received',
        'request_to_send_received return_code'
  say; say 'Routine called:  CMRCV'
  select
    when (return_code = CM_OK) then
      do
        call TraceParms 'data_received status_received'
        if (data_received ¬= CM_NO_DATA_RECEIVED) then
          do
            receive_buffer = left(receive_buffer,received_length)
            complete_line = complete_line || receive_buffer
          end
        if (data_received = CM_COMPLETE_DATA_RECEIVED) then
          do
            /*-------------------------------------------------------*/
            /* Use EXECIO to write the data to OUTPUT LOGFILE A      */
            /* and reset the complete_line variable to nulls.        */
            /*-------------------------------------------------------*/
            address command 'EXECIO 1 DISKW OUTPUT LOGFILE A (FINIS',
                            'STRING' complete_line
            complete_line = ''
          end
        /*-------------------------------------------------------*/
        /* Determine whether a confirmation request has been     */
        /* received.  If so, respond with a positive reply.      */
        /*-------------------------------------------------------*/
        if (status_received = CM_CONFIRM_RECEIVED) |,
```

```
           (status_received = CM_CONFIRM_SEND_RECEIVED) |,
           (status_received = CM_CONFIRM_DEALLOC_RECEIVED) then
          do
            /*-------------------------------------------------------*/
            /* Issue Confirmed to reply to the partner.              */
            /*-------------------------------------------------------*/
            'CMCFMD conversation_ID return_code'
            say; say 'Routine called:  CMCFMD'
            if (return_code ¬= CM_OK) then call ErrorHandler 'CMCFMD'
            call TraceParms
          end
      end
    otherwise
      call ErrorHandler 'CMRCV'
  end
end
/*---------------------------------------------------------------------*/
/* Deallocate the conversation normally.                               */
/*---------------------------------------------------------------------*/
'CMDEAL conversation_ID return_code'
say; say 'Routine called:  CMDEAL'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMDEAL'
call TraceParms

GetOut:
  exit

/*-------------------------- Subroutines --------------------------*/

TraceParms:
/*---------------------------------------------------------------------*/
/* Display parameters and their values as passed to this subroutine.*/
/*---------------------------------------------------------------------*/
parse arg parmlist
do word_num = 1 to words(parmlist)
  parameter = word(parmlist,word_num)
  select
    when (parameter = 'return_code') then
      say '  return_code is' cm_return_code.return_code
    when (parameter = 'buffer') then
      say '  buffer is' left(buffer,send_length)
    when (parameter = 'receive_buffer') then
      say '  buffer is' left(receive_buffer,received_length)
    when (parameter = 'data_received') then
      say '  data_received is' cm_data_received.data_received
    when (parameter = 'status_received') then
      say '  status_received is' cm_status_received.status_received
    when (parameter = 'request_to_send_received') then
      say '  request_to_send_received is',
          cm_request_to_send_received.request_to_send_received
    when (parameter = 'sync_level') then
      say '  sync_level is' cm_sync_level.sync_level
    when (parameter = 'prepare_to_receive_type') then
      say '  prepare_to_receive_type is',
          cm_prepare_to_receive_type.prepare_to_receive_type
    when (parameter = 'deallocate_type') then
      say '  deallocate_type is' cm_deallocate_type.deallocate_type
    when (parameter = 'conversation_security_type') then
      say '  conversation_security_type is',
          xc_conversation_security_type.conversation_security_type
    otherwise
      say ' ' parameter 'is' value(parameter)
  end
end
/*---------------------------------------------------------------------*/
/* Extract the current conversation state of the local program.        */
/*---------------------------------------------------------------------*/
```

```
/* Commenting out next four lines ...
'CMECS conversation_ID conversation_state return_code'
if (return_code = CM_OK) then
  say '  conversation_state is =>',
      cm_conversation_state.conversation_state
... */

return


Error:
/*-------------------------------------------------------------------*/
/* Report error when REXX special variable RC is not 0.              */
/*-------------------------------------------------------------------*/
say
say '* ERROR:  REXX has detected an error'
say '          The return code variable RC was set to' rc
call AbnormalEnd
signal GetOut


ErrorHandler:
/*-------------------------------------------------------------------*/
/* Report routine that failed and the error return code.             */
/*-------------------------------------------------------------------*/
parse arg routine_name
say
say '* ERROR:  An error occurred during a' routine_name 'call'
say '          The return_code was set to' cm_return_code.return_code
call AbnormalEnd
signal GetOut

AbnormalEnd:
/*-------------------------------------------------------------------*/
/* Abnormally deallocate the conversation.  Since we are exiting     */
/* due to an error, we will not display an error message if the      */
/* Set_Deallocate_Type or Deallocate call encounters an error.       */
/*-------------------------------------------------------------------*/
deallocate_type = CM_DEALLOCATE_ABEND
'CMSDT conversation_ID deallocate_type return_code'
say; say 'Routine called:  CMSDT'
if (return_code = CM_OK) then
  do
    call TraceParms
    'CMDEAL conversation_ID return_code'
    say; say 'Routine called:  CMDEAL'
    if (return_code = CM_OK) then
      call TraceParms
  end

return
```

The SENDBACK Sample Intermediate Server Exec

```
/*===================================================================*/
/*  SENDBACK EXEC - Sample intermediate server application.          */
/*===================================================================*/

arg resource_ID .          /* :nick. value from $SERVER$ NAMES file */
/*-------------------------------------------------------------------*/
/* Set up REXX environment for program-to-program communications     */
/* and enable error trapping of REXX errors.                         */
/*-------------------------------------------------------------------*/
address cpicomm
signal on error
/*-------------------------------------------------------------------*/
/* Equate pseudonyms to their integer values based on the            */
/* definitions contained in the CMREXX COPY file.                    */
```

```
/*-------------------------------------------------------------------*/
address command 'EXECIO * DISKR CMREXX COPY * (FINIS STEM PSEUDONYM.'
do index = 1 to pseudonym.0
  interpret pseudonym.index
end
/*-------------------------------------------------------------------*/
/* Identify the application as manager of the private resource.      */
/* Remember the resource_ID value for later use in XCTRRM by         */
/* storing it in save_res_ID.                                        */
/*-------------------------------------------------------------------*/
resource_ID = word(resource_ID 'GET',1)
save_res_ID = resource_ID
resource_manager_type = XC_PRIVATE
service_mode = XC_SEQUENTIAL
security_level_flag = XC_REJECT_SECURITY_NONE
'XCIDRM resource_ID resource_manager_type service_mode',
       'security_level_flag return_code'
say; say 'Routine called:  XCIDRM'
if (return_code ¬= CM_OK) then
  do
    say
    say '* ERROR:  An error occurred during an XCIDRM call'
    say '          The return_code was set to',
        cm_return_code.return_code
    signal GetOut
  end
call TraceParms
/*-------------------------------------------------------------------*/
/* Start continuous Wait_on_Event loop.                              */
/* Any console input will end the loop.                              */
/*-------------------------------------------------------------------*/
send_data = ''
do forever
  say; say 'Waiting for an event to occur.  Enter "QUIT" to exit.'
  /*-----------------------------------------------------------------*/
  /* Issue Wait_on_Event to wait for the next event to occur.        */
  /*-----------------------------------------------------------------*/
  'XCWOE resource_ID conversation_ID event_type event_info_length',
        'event_buffer return_code'
  say; say 'Routine called:  XCWOE'
  if (return_code ¬= CM_OK) then call ErrorHandler 'XCWOE'
  call TraceParms 'conversation_ID event_type'
  /*-----------------------------------------------------------------*/
  /* Choose next action based on type of event.                      */
  /*-----------------------------------------------------------------*/
  select
    when (event_type = XC_ALLOCATION_REQUEST) then
      do
        call AcceptConv
        call StartConv
      end
    when (event_type = XC_INFORMATION_INPUT) then
      call ReceiveInfo
    when (event_type = XC_CONSOLE_INPUT) then
      /*---------------------------------------------------------------*/
      /* Leave the Wait_on_Event loop.                                 */
      /*---------------------------------------------------------------*/
      leave
    otherwise
      do
        say
        say '* ERROR:  Wait_on_Event reported event_type',
            xc_event_type.event_type
      end
  end /* select */
  /*-----------------------------------------------------------------*/
  /* When notice of partner's deallocation is received, leave the    */
```

```
  /* Wait_on_Event loop.                                              */
  /*------------------------------------------------------------------*/
  /* Commenting out next three lines ...
  if (return_code = CM_DEALLOCATED_NORMAL) |,
     (status_received = CM_CONFIRM_DEALLOC_RECEIVED) then
    leave
  ... */
end /* do forever */

GetOut:
  call TerminateRes
  exit

/*------------------------- Subroutines --------------------------*/

AcceptConv:
/*------------------------------------------------------------------*/
/* Accept the incoming conversation.                                */
/* Store the conversation identifier in save_con_ID1.               */
/*------------------------------------------------------------------*/
'CMACCP conversation_ID return_code'
save_con_ID1 = conversation_ID
say; say 'Routine called:  CMACCP'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMACCP'
call TraceParms 'conversation_ID'
/*------------------------------------------------------------------*/
/* Extract conversation_type to ensure the conversation is mapped.  */
/*------------------------------------------------------------------*/
'CMECT conversation_ID conversation_type return_code'
say; say 'Routine called:  CMECT'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMECT'
call TraceParms
/*------------------------------------------------------------------*/
/* If the conversation is basic, deallocate abnormally.             */
/*------------------------------------------------------------------*/
if (conversation_type = CM_BASIC_CONVERSATION) then
  do
    say; say '* ERROR:  Accepting and deallocating a basic',
             'conversation'
    /*--------------------------------------------------------------*/
    /* Call Send_Error to notify partner that error was detected.   */
    /* Since the program is going to exit, do not check the         */
    /* Send_Error results for an error.                             */
    /*--------------------------------------------------------------*/
    'CMSERR conversation_ID request_to_send_received return_code'
    say; say 'Routine called:  CMSERR'
    if (return_code = CM_OK) then
      call TraceParms
    call AbnormalEnd
    signal GetOut
  end
/*------------------------------------------------------------------*/
/* Extract the access security user ID for the conversation.        */
/*------------------------------------------------------------------*/
'XCECSU conversation_ID security_user_ID security_user_ID_length',
       'return_code'
say; say 'Routine called:  XCECSU'
if (return_code ¬= CM_OK) then call ErrorHandler 'XCECSU'
call TraceParms 'security_user_ID'

return


StartConv:
/*------------------------------------------------------------------*/
/* StartConv will establish a conversation with the resource        */
/* manager on behalf of the file requester.                         */
```

```
/* First, initialize the conversation to the resource manager.      */
/*------------------------------------------------------------------*/
sym_dest_name = 'GETFILE'
'CMINIT conversation_ID sym_dest_name return_code'
save_con_ID2 = conversation_ID
say; say 'Routine called:  CMINIT'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMINIT'
call TraceParms 'conversation_ID'
/*------------------------------------------------------------------*/
/* Pass the requesting program's access user ID to resource manager.*/
/*------------------------------------------------------------------*/
client_user_ID = security_user_ID
'XCSCUI conversation_ID client_user_ID return_code'
say; say 'Routine called:  XCSCUI'
if (return_code ¬= CM_OK) then call ErrorHandler 'XCSCUI'
call TraceParms 'client_user_ID'
/*------------------------------------------------------------------*/
/* Allocate conversation. Conversation_ID still equals save_con_ID2.*/
/*------------------------------------------------------------------*/
'CMALLC conversation_ID return_code'
say; say 'Routine called:  CMALLC'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMALLC'
call TraceParms

return


ReceiveInfo:
/*------------------------------------------------------------------*/
/* Start a Receive loop.                                            */
/* Receive data, status, or both from conversation partner.         */
/*------------------------------------------------------------------*/
requested_length = event_info_length
do until (data_received = CM_COMPLETE_DATA_RECEIVED) |,
         (data_received = CM_NO_DATA_RECEIVED)
  'CMRCV conversation_ID receive_buffer requested_length',
        'data_received received_length status_received',
        'request_to_send_received return_code'
  CMRCV_return_code = return_code
  say; say 'Routine called:  CMRCV'
  select
    when (CMRCV_return_code = CM_OK) then
      do
        call TraceParms 'data_received status_received'
        if (data_received ¬= CM_NO_DATA_RECEIVED) then
          do
            receive_buffer = left(receive_buffer,received_length)
            send_data = send_data || receive_buffer
          end
        /*----------------------------------------------------------*/
        /* Determine whether a confirmation request has been        */
        /* received.  If so, respond with a positive reply.         */
        /*----------------------------------------------------------*/
        if (status_received = CM_CONFIRM_RECEIVED) |,
           (status_received = CM_CONFIRM_SEND_RECEIVED) |,
           (status_received = CM_CONFIRM_DEALLOC_RECEIVED) then
          do
            /*--------------------------------------------------*/
            /* Issue Confirmed to reply to the partner.         */
            /*--------------------------------------------------*/
            'CMCFMD conversation_ID return_code'
            say; say 'Routine called:  CMCFMD'
            if (return_code ¬= CM_OK) then call ErrorHandler 'CMCFMD'
            call TraceParms
          end
        if (data_received = CM_COMPLETE_DATA_RECEIVED) then
          /*--------------------------------------------------*/
```

```
          /* Forward data to partner on the other conversation.     */
          /*-------------------------------------------------------*/
          call SendInfo
        if (status_received = CM_SEND_RECEIVED) |,
           (status_received = CM_CONFIRM_SEND_RECEIVED) then
          /*-------------------------------------------------------*/
          /* The server should only get send control when one      */
          /* partner has completed sending data.                   */
          /*-------------------------------------------------------*/
          call PrepReceive
        else
          if (status_received = CM_CONFIRM_DEALLOC_RECEIVED) then
            do
              say; say 'Conversation deallocated by requester'
              call EndConv
            end
      end
    when (CMRCV_return_code = CM_DEALLOCATED_NORMAL) then
      do
        call TraceParms 'data_received status_received'
        say; say 'Conversation deallocated by requester'
        call EndConv
      end
    otherwise
      call ErrorHandler 'CMRCV'
  end
end

return


SendInfo:
/*--------------------------------------------------------------------*/
/* Send data received on one conversation to partner on other         */
/* conversation.  The send_data variable contains either the name     */
/* of the requested file or a line from the file, and it was set in   */
/* ReceiveInfo.  Conversation_ID was last set on CMRCV call.          */
/* Reset it to the ID of the other conversation.                      */
/*--------------------------------------------------------------------*/
if (conversation_ID = save_con_ID1) then
  con_ID = save_con_ID2
else
  con_ID = save_con_ID1
buffer = send_data
send_length = length(buffer)
'CMSEND con_ID buffer send_length',
       'request_to_send_received return_code'
say; say 'Routine called:  CMSEND'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMSEND'
call TraceParms 'con_ID'
send_data = ''           /* reset received data variable to nulls */

return


PrepReceive:
/*--------------------------------------------------------------------*/
/* When send control is received on one conversation, the             */
/* intermediate server is ready to transfer send control to           */
/* partner on the other conversation.                                 */
/*--------------------------------------------------------------------*/
if (conversation_ID = save_con_ID1) then
  con_ID = save_con_ID2
else
  con_ID = save_con_ID1
'CMPTR con_ID return_code'
say; say 'Routine called:  CMPTR'
```

```
if (return_code ¬= CM_OK) then call ErrorHandler 'CMPTR'
call TraceParms 'con_ID'

return


TraceParms:
/*-------------------------------------------------------------------*/
/* Display parameters and their values as passed to this subroutine.*/
/*-------------------------------------------------------------------*/
parse arg parmlist
do word_num = 1 to words(parmlist)
  parameter = word(parmlist,word_num)
  select
    when (parameter = 'return_code') then
      say '  return_code is' cm_return_code.return_code
    when (parameter = 'buffer') then
      say '  buffer is' left(buffer,send_length)
    when (parameter = 'receive_buffer') then
      say '  buffer is' left(receive_buffer,received_length)
    when (parameter = 'data_received') then
      say '  data_received is' cm_data_received.data_received
    when (parameter = 'status_received') then
      say '  status_received is' cm_status_received.status_received
    when (parameter = 'request_to_send_received') then
      say '  request_to_send_received is',
          cm_request_to_send_received.request_to_send_received
    when (parameter = 'send_type') then
      say '  send_type is' cm_send_type.send_type
    when (parameter = 'deallocate_type') then
      say '  deallocate_type is' cm_deallocate_type.deallocate_type
    when (parameter = 'conversation_type') then
      say '  conversation_type is',
          cm_conversation_type.conversation_type
    when (parameter = 'resource_manager_type') then
      say '  resource_manager_type is',
          xc_resource_manager_type.resource_manager_type
    when (parameter = 'service_mode') then
      say '  service_mode is' xc_service_mode.service_mode
    when (parameter = 'security_level_flag') then
      say '  security_level_flag is',
          xc_security_level_flag.security_level_flag
    when (parameter = 'event_type') then
      say '  event_type is' xc_event_type.event_type
    otherwise
      say ' ' parameter 'is' value(parameter)
  end
end
/*-------------------------------------------------------------------*/
/* Extract the current conversation state of the local program.      */
/*-------------------------------------------------------------------*/
/* Commenting out next four lines ...
'CMECS conversation_ID conversation_state return_code'
if (return_code = CM_OK) then
  say '  conversation_state is =>',
      cm_conversation_state.conversation_state
... */

return


EndConv:
/*-------------------------------------------------------------------*/
/* Deallocate the conversation with the resource manager.            */
/*-------------------------------------------------------------------*/
conversation_ID = save_con_ID2
'CMDEAL conversation_ID return_code'
```

```
say; say 'Routine called:  CMDEAL'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMDEAL'
call TraceParms 'conversation_ID'

return


Error:
/*-------------------------------------------------------------------*/
/* Report error when REXX special variable RC is not 0.              */
/*-------------------------------------------------------------------*/
say
say '* ERROR:  REXX has detected an error'
say '          The return code variable RC was set to' rc
call AbnormalEnd
signal GetOut


ErrorHandler:
/*-------------------------------------------------------------------*/
/* Report routine that failed and the error return code.             */
/*-------------------------------------------------------------------*/
parse arg routine_name
say
say '* ERROR:  An error occurred during a' routine_name 'call'
say '          The return_code was set to' cm_return_code.return_code
call AbnormalEnd
signal GetOut


AbnormalEnd:
/*-------------------------------------------------------------------*/
/* Abnormally deallocate the conversation.  Since we are exiting     */
/* due to an error, we will not display an error message if the      */
/* Set_Deallocate_Type or Deallocate call encounters an error.       */
/* Use conversation IDs in save_con_ID1 and save_con_ID2.            */
/*-------------------------------------------------------------------*/
deallocate_type = CM_DEALLOCATE_ABEND
conversation_ID = save_con_ID
'CMSDT conversation_ID deallocate_type return_code'
say; say 'Routine called:  CMSDT'
if (return_code = CM_OK) then
  do
    call TraceParms
    'CMDEAL conversation_ID return_code'
    say; say 'Routine called:  CMDEAL'
    if (return_code = CM_OK) then
      call TraceParms
  end
conversation_ID = save_con_ID2
'CMSDT conversation_ID deallocate_type return_code'
say; say 'Routine called:  CMSDT'
if (return_code = CM_OK) then
  do
    'CMDEAL conversation_ID return_code'
    say; say 'Routine called:  CMDEAL'
  end

return


TerminateRes:
/*-------------------------------------------------------------------*/
/* TerminateRes will terminate ownership of the specified resource. */
/* Use resource name stored in save_res_ID at start of program.      */
/*-------------------------------------------------------------------*/
resource_ID = save_res_ID
```

```
'XCTRRM resource_ID return_code'
say; say 'Routine called:  XCTRRM'
if (return_code ¬= CM_OK) then
  do
    say
    say '* ERROR:  An error occurred during an XCTRRM call'
    say '          The return_code was set to',
        cm_return_code.return_code
  end
else
  call TraceParms

return
```

The SENDSERV Sample Resource Manager Exec

```
/*====================================================================*/
/*  SENDSERV EXEC - Sample server application.                        */
/*====================================================================*/

arg resource_ID .          /* :nick. value from $SERVER$ NAMES file */
/*--------------------------------------------------------------------*/
/* Set up REXX environment for program-to-program communications    */
/* and enable error trapping of REXX errors.                         */
/*--------------------------------------------------------------------*/
address cpicomm
signal on error
/*--------------------------------------------------------------------*/
/* Equate pseudonyms to their integer values based on the           */
/* definitions contained in the CMREXX COPY file.                    */
/*--------------------------------------------------------------------*/
address command 'EXECIO * DISKR CMREXX COPY * (FINIS STEM PSEUDONYM.'
do index = 1 to pseudonym.0
  interpret pseudonym.index
end
/*--------------------------------------------------------------------*/
/* Identify the application as manager of the private resource.     */
/* Remember the resource_ID value for later use in XCTRRM by        */
/* storing it in save_res_ID.                                        */
/*--------------------------------------------------------------------*/
resource_ID = word(resource_ID 'GET',1)
save_res_ID = resource_ID
resource_manager_type = XC_PRIVATE
service_mode = XC_SEQUENTIAL
security_level_flag = XC_REJECT_SECURITY_NONE
'XCIDRM resource_ID resource_manager_type service_mode',
       'security_level_flag return_code'
say; say 'Routine called:  XCIDRM'
if (return_code ¬= CM_OK) then
  do
    say
    say '* ERROR:  An error occurred during an XCIDRM call'
    say '          The return_code was set to',
        cm_return_code.return_code
    signal GetOut
  end
call TraceParms
/*--------------------------------------------------------------------*/
/* Start continuous Wait_on_Event loop.                              */
/* Any console input will end the loop.                              */
/*--------------------------------------------------------------------*/
requested_file = ''
do forever
  say; say 'Waiting for an event to occur.  Enter "QUIT" to exit.'
  /*------------------------------------------------------------------*/
  /* Issue Wait_on_Event to wait for the next event to occur.       */
  /*------------------------------------------------------------------*/
```

```
  'XCWOE resource_ID conversation_ID event_type event_info_length',
        'event_buffer return_code'
  say; say 'Routine called:  XCWOE'
  if (return_code ¬= CM_OK) then call ErrorHandler 'XCWOE'
  call TraceParms
  /*-------------------------------------------------------------------*/
  /* Choose next action based on type of event.                        */
  /*-------------------------------------------------------------------*/
  select
    when (event_type = XC_ALLOCATION_REQUEST) then
      call AcceptConv
    when (event_type = XC_INFORMATION_INPUT) then
      call ReceiveInfo
    when (event_type = XC_CONSOLE_INPUT) then
      /*---------------------------------------------------------------*/
      /* Leave the Wait_on_Event loop.                                 */
      /*---------------------------------------------------------------*/
      leave
    otherwise
      do
        say
        say '* ERROR:  Wait_on_Event reported event_type',
            xc_event_type.event_type
      end
  end /* select */
  /*-------------------------------------------------------------------*/
  /* When notice of partner's deallocation is received, leave the      */
  /* Wait_on_Event loop.                                               */
  /*-------------------------------------------------------------------*/
  if (return_code = CM_DEALLOCATED_NORMAL) |,
     (status_received = CM_CONFIRM_DEALLOC_RECEIVED) then
    leave
end /* do forever */

GetOut:
  call TerminateRes
  exit

/*------------------------- Subroutines ----------------------------*/

AcceptConv:
/*-------------------------------------------------------------------*/
/* Accept the incoming conversation.                                 */
/* Store the conversation identifier in save_con_ID.                 */
/*-------------------------------------------------------------------*/
'CMACCP conversation_ID return_code'
save_con_ID = conversation_ID
say; say 'Routine called:  CMACCP'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMACCP'
call TraceParms
/*-------------------------------------------------------------------*/
/* Extract conversation_type to ensure the conversation is mapped.   */
/*-------------------------------------------------------------------*/
'CMECT conversation_ID conversation_type return_code'
say; say 'Routine called:  CMECT'
if (return_code ¬= CM_OK) then call ErrorHandler 'CMECT'
call TraceParms
/*-------------------------------------------------------------------*/
/* If the conversation is basic, deallocate abnormally.              */
/*-------------------------------------------------------------------*/
if (conversation_type = CM_BASIC_CONVERSATION) then
  do
    say; say '* ERROR:  Accepting and deallocating a basic',
             'conversation'
    /*---------------------------------------------------------------*/
    /* Call Send_Error to notify partner that error was detected.    */
    /* Since the program is going to exit, do not check the          */
```

```
    /* Send_Error results for an error.                          */
    /*-----------------------------------------------------------*/
    'CMSERR conversation_ID request_to_send_received return_code'
    say; say 'Routine called:  CMSERR'
    if (return_code = CM_OK) then
      call TraceParms
    call AbnormalEnd
    signal GetOut
  end
/*-------------------------------------------------------------------*/
/* Extract the access security user ID for the conversation.         */
/*-------------------------------------------------------------------*/
'XCECSU conversation_ID security_user_ID security_user_ID_length',
       'return_code'
say; say 'Routine called:  XCECSU'
if (return_code ¬= CM_OK) then call ErrorHandler 'XCECSU'
call TraceParms 'security_user_ID'

return


ReceiveInfo:
/*-------------------------------------------------------------------*/
/* Start a Receive loop.                                             */
/* Receive data, status, or both from conversation partner.          */
/*-------------------------------------------------------------------*/
requested_length = event_info_length
do until (data_received = CM_COMPLETE_DATA_RECEIVED) |,
         (data_received = CM_NO_DATA_RECEIVED)
  'CMRCV conversation_ID receive_buffer requested_length',
        'data_received received_length status_received',
        'request_to_send_received return_code'
  CMRCV_return_code = return_code
  say; say 'Routine called:  CMRCV'
  select
    when (CMRCV_return_code = CM_OK) then
      do
        call TraceParms 'data_received status_received'
        if (data_received ¬= CM_NO_DATA_RECEIVED) then
          do
            receive_buffer = left(receive_buffer,received_length)
            requested_file = requested_file || receive_buffer
          end
        /*-----------------------------------------------------------*/
        /* Determine whether a confirmation request has been         */
        /* received.  If so, respond with a positive reply.          */
        /*-----------------------------------------------------------*/
        if (status_received = CM_CONFIRM_RECEIVED) |,
           (status_received = CM_CONFIRM_SEND_RECEIVED) |,
           (status_received = CM_CONFIRM_DEALLOC_RECEIVED) then
          do
            /*---------------------------------------------------*/
            /* Issue Confirmed to reply to the partner.          */
            /*---------------------------------------------------*/
            'CMCFMD conversation_ID return_code'
            say; say 'Routine called:  CMCFMD'
            if (return_code ¬= CM_OK) then call ErrorHandler 'CMCFMD'
            call TraceParms
          end
        if (status_received = CM_SEND_RECEIVED) |,
           (status_received = CM_CONFIRM_SEND_RECEIVED) then
          call SendFile
        else
          if (status_received = CM_CONFIRM_DEALLOC_RECEIVED) then
            do
              say; say 'Conversation deallocated by partner'
              requested_file = ''
```

```
            end
      end
    when (CMRCV_return_code = CM_DEALLOCATED_NORMAL) then
      do
        call TraceParms 'data_received status_received'
        say; say 'Conversation deallocated by partner'
        requested_file = ''
      end
    otherwise
      call ErrorHandler 'CMRCV'
  end
end

return


SendFile:
/*--------------------------------------------------------------------*/
/* Read the contents of the requested file and send each line of    */
/* the file to the partner program.                                   */
/*--------------------------------------------------------------------*/
address command 'EXECIO * DISKR' requested_file '(FINIS STEM LINE.'
do index = 1 to line.0
  if (index = line.0) then
    /*------------------------------------------------------------------*/
    /* Reset the send_type conversation characteristic just         */
    /* before the final Send_Data call.                                 */
    /*------------------------------------------------------------------*/
    do
      send_type = CM_SEND_AND_PREP_TO_RECEIVE
      'CMSST conversation_ID send_type return_code'
      say; say 'Routine called:  CMSST'
      if (return_code ¬= CM_OK) then call ErrorHandler 'CMSST'
      call TraceParms
    end
  buffer = line.index
  send_length = length(buffer)
  'CMSEND conversation_ID buffer send_length',
         'request_to_send_received return_code'
  say; say 'Routine called:  CMSEND'
  if (return_code ¬= CM_OK) then call ErrorHandler 'CMSEND'
  call TraceParms
end

return


TraceParms:
/*--------------------------------------------------------------------*/
/* Display parameters and their values as passed to this subroutine.*/
/*--------------------------------------------------------------------*/
parse arg parmlist
do word_num = 1 to words(parmlist)
  parameter = word(parmlist,word_num)
  select
    when (parameter = 'return_code') then
      say '  return_code is' cm_return_code.return_code
    when (parameter = 'buffer') then
      say '  buffer is' left(buffer,send_length)
    when (parameter = 'receive_buffer') then
      say '  buffer is' left(receive_buffer,received_length)
    when (parameter = 'data_received') then
      say '  data_received is' cm_data_received.data_received
    when (parameter = 'status_received') then
      say '  status_received is' cm_status_received.status_received
    when (parameter = 'request_to_send_received') then
      say '  request_to_send_received is',
```

```
          cm_request_to_send_received.request_to_send_received
      when (parameter = 'send_type') then
        say '  send_type is' cm_send_type.send_type
      when (parameter = 'deallocate_type') then
        say '  deallocate_type is' cm_deallocate_type.deallocate_type
      when (parameter = 'conversation_type') then
        say '  conversation_type is',
            cm_conversation_type.conversation_type
      when (parameter = 'resource_manager_type') then
        say '  resource_manager_type is',
            xc_resource_manager_type.resource_manager_type
      when (parameter = 'service_mode') then
        say '  service_mode is' xc_service_mode.service_mode
      when (parameter = 'security_level_flag') then
        say '  security_level_flag is',
            xc_security_level_flag.security_level_flag
      when (parameter = 'event_type') then
        say '  event_type is' xc_event_type.event_type
      otherwise
        say ' ' parameter 'is' value(parameter)
    end
end
/*--------------------------------------------------------------------*/
/* Extract the current conversation state of the local program.       */
/*--------------------------------------------------------------------*/
/* Commenting out next four lines ...
'CMECS conversation_ID conversation_state return_code'
if (return_code = CM_OK) then
  say '  conversation_state is =>',
      cm_conversation_state.conversation_state
... */

return


Error:
/*--------------------------------------------------------------------*/
/* Report error when REXX special variable RC is not 0.               */
/*--------------------------------------------------------------------*/
say
say '* ERROR:  REXX has detected an error'
say '          The return code variable RC was set to' rc
call AbnormalEnd
signal GetOut


ErrorHandler:
/*--------------------------------------------------------------------*/
/* Report routine that failed and the error return code.              */
/*--------------------------------------------------------------------*/
parse arg routine_name
say
say '* ERROR:  An error occurred during a' routine_name 'call'
say '          The return_code was set to' cm_return_code.return_code
call AbnormalEnd
signal GetOut


AbnormalEnd:
/*--------------------------------------------------------------------*/
/* Abnormally deallocate the conversation.  Since we are exiting      */
/* due to an error, we will not display an error message if the       */
/* Set_Deallocate_Type or Deallocate call encounters an error.        */
/* Use conversation ID in save_con_ID, from start of conversation.    */
/*--------------------------------------------------------------------*/
deallocate_type = CM_DEALLOCATE_ABEND
conversation_ID = save_con_ID
```

```
'CMSDT conversation_ID deallocate_type return_code'
say; say 'Routine called:  CMSDT'
if (return_code = CM_OK) then
  do
    call TraceParms
    'CMDEAL conversation_ID return_code'
    say; say 'Routine called:  CMDEAL'
    if (return_code = CM_OK) then
      call TraceParms
  end

return


TerminateRes:
/*--------------------------------------------------------------------*/
/* TerminateRes will terminate ownership of the specified resource. */
/* Use resource name stored in save_res_ID at start of program.     */
/*--------------------------------------------------------------------*/
resource_ID = save_res_ID
'XCTRRM resource_ID return_code'
say; say 'Routine called:  XCTRRM'
if (return_code ¬= CM_OK) then
  do
    say
    say '* ERROR:  An error occurred during an XCTRRM call'
    say '          The return_code was set to',
        cm_return_code.return_code
  end
else
  call TraceParms

return
```

Conclusion

In this chapter we expanded our use of CPI Communications routines to include the VM extensions, we learned about intermediate servers and we successfully changed security information for the conversation.

As we pointed out in the introduction, this book is intended to be merely an introduction to SAA CPI Communications on VM and to the VM extensions to CPI Communications. There is still more for you to learn. The *Common Programming Interface Communications Reference* and the *z/VM: CMS Application Development Guide* contain additional sample programs and more information on using CPI Communications. In addition, the *z/VM: Connectivity* book contains information that can help you set up your virtual machine to manage system resources, communicate with resources, and communicate with other programs through a TSAF collection or an SNA network.

Appendix A. Event Management for CPI Communications

The Wait_on_Event (XCWOE) call is provided by z/VM to allow an application to receive notification of various communications events. We introduced this call earlier in our example programs.

It is important to understand that Wait_on_Event causes an entire virtual machine to be blocked until the call completes. So, a better choice for multitasking applications is to use CMS event management services. These services allow an application to wait for events while blocking just a single thread. Hence, other threads can continue to perform work while there is a wait outstanding.

The occurrences of CPI Communications events are represented by a system event called VMCPIC. By reporting information through this event, CPI Communications allows an application to use all the facilities of event management services to monitor and respond to these conditions, with the additional benefit of avoiding undue serialization in multitasking applications.

The following types of events are reported by the VMCPIC system event:
- Allocation requests
- Information input from partner
- Resource revoked notification.

Note: When using the VMCPIC event to monitor allocation requests or information input, these events are reported only once, whereas Wait_on_Event reports these events continuously until the appropriate action is taken.

Console input and user events also can be handled through the use of event management services. CMS provides a system event, called VMCONINPUT, that allows an application to monitor console input events. Additionally, an application can use the event management functions to generate and process its own user events.

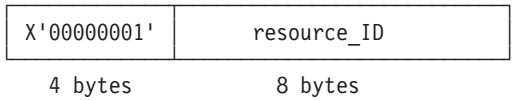
The VMCPIC System Event

When a VMCPIC system event is signalled, data comparable to the information provided by the Wait_on_Event call is associated with the signal. A portion of this event data composes the event key.

The following list contains a description of the event data available when a VMCPIC event is signalled:

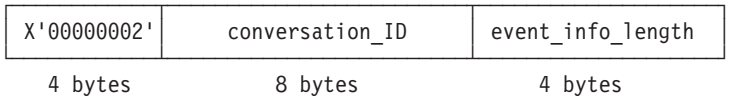
- Allocation requests:

  The event data associated with a signal for an allocation request consists of X'00000001' concatenated with the *resource_ID*:



- Information input:

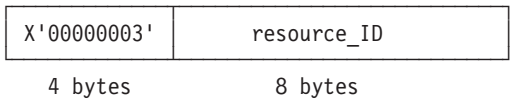
  The event data associated with a signal for information input consists of X'00000002' concatenated with the *conversation_ID* concatenated with the *event_info_length*:



Note: The *event_info_length* may be greater than the actual data sent by your remote partner. This information is part of the mapped conversation data record built and sent by CPI Communications at your remote partner.

- Resource revoked notification:

  The event data associated with a signal for a resource revoked notification also consists of X'00000003' concatenated with the *resource_ID*:



For example, let's take a look at the event data that would be associated with an information input event type. If our communications partner on conversation ID 00000000 sent us the data string "Begin transaction" on a mapped conversation, the VMCPIC event signal would have associated with it the following hexadecimal representation of the event data:

X'00000002F0F0F0F0F0F0F0F000000015'

The first 4 bytes of the event data indicate that the *event_type* is 2, or XC_INFORMATION_INPUT as we refer to it in this book. The next 8 bytes hold the *conversation_ID* of 00000000. And, the final 4 bytes inform us of the length of information that is available for receipt.

Managing Events

CMS declares the event services external functions, constants, and return and reason codes in a series of programming language binding files. The APILOAD command is provided for processing these files in a REXX application.

Including VMREXMT provides us with all of the REXX binding files. (Note that it is possible to be more selective as to which files get included, but the safe way when starting out is to include all of them.)

The following lines of code will accomplish this task:

```
/*--------------------------------------------------------------------*/
/* Process REXX binding files.                                        */
/*--------------------------------------------------------------------*/
'APILOAD (VMREXMT)'
```

The approach to waiting on events involves first creating an event monitor through the EventMonitorCreate function. This tells event management services which event your program is interested in and if you want to be notified only of those events of a certain type.

To monitor CPI Communications events, we will need to create an event monitor for the VMCPIC system event. And, to be able to handle console input, we will also want to monitor the VMCONINPUT system event.

Here is an example illustrating one way to do it:

```
   ⋮
/*-------------------------------------------------------------------*/
/* Create an event monitor for the VMCPIC event, specifying a       */
/* wildcard key of '*' so all VMCPIC events will match.  Also       */
/* monitor the VMCONINPUT event to trap console input.              */
/*-------------------------------------------------------------------*/
/* Indicate monitor should persist until EventMonitorDeleted or     */
/*   the process is terminated.                                     */
monitor_flag.1 = vm_evn_no_auto_delete

/* Indicate all threads in the process containing the monitor       */
/*   remain dispatchable.                                           */
monitor_flag.2 = vm_evn_async_monitor

/* Indicate any loose signals should be bound to the monitor.       */
monitor_flag.3 = vm_evn_bind_loose_signals

monitor_flag_size = 3       /* Monitor_flag composed of 3 elements  */

number_of_events = 2        /* Monitor 2 events: VMCPIC, VMCONINPUT */

/* Specify the names of the events to monitor.                      */
event_name_address.1 = 'VMCPIC'
event_name_address.2 = 'VMCONINPUT'

event_name_length.1 = length(event_name_address.1)
event_name_length.2 = length(event_name_address.2)

/* Use a wildcard key of '*' to match any occurrence.               */
event_key_address.1 = '*'; event_key_length.1 = 1
event_key_address.2 = '*'; event_key_length.2 = 1

/* Allow the list of bound signals to grow without limit.           */
bound_signal_limit.1 = -1; bound_signal_limit.2 = -1

/* The monitored condition is satisfied if one of the event list    */
/* entries is signalled.                                            */
event_count = 1

call CSL 'EventMonitorCreate retcode reascode monitor_token',
         'monitor_flag monitor_flag_size number_of_events',
         'event_name_address event_name_length event_key_address',
         'event_key_length bound_signal_limit event_count'
   ⋮
```

Next, we issue EventWait to actually enter a wait for the occurrence of either the VMCPIC or the VMCONINPUT event. Typically, the call to EventWait is placed inside a loop, and an application that is waiting for an event would generally want to keep its end of a conversation in **Receive** state.

Here is an example of the EventWait call:

```
   ⋮
/*-------------------------------------------------------------------*/
/* Issue EventWait to wait for the next event to occur.             */
/*-------------------------------------------------------------------*/
call CSL 'EventWait retcode reascode monitor_token',
         'number_of_events event_flag'
   ⋮
```

If the EventWait completes due to the signalling of a VMCPIC event, we will want to call EventRetrieve to obtain the associated event data to determine which *event_type* occurred. For example, upon learning that a VMCPIC information input event has been signalled, we will want to receive the information our communications partner sent.

When EventWait completes because the VMCONINPUT event was signalled, we will need to read a line from the terminal input buffer.

Remember, if we create a monitor for more than one event, it is possible for an EventWait call to complete with an indication that several of the events have been signalled.

We might use a code fragment like the following one:

```
  :
/*--------------------------------------------------------------------*/
/* An event has occurred, so get the data describing the event.       */
/*--------------------------------------------------------------------*/
/* Check each of the events we are monitoring.                        */
do index = 1 to event_flag.0

  /* Check if this event has been signalled.                          */
  if event_flag.index >= 0 then
    do
      if event_name_address.index = 'VMCONINPUT' then
        do
          /* Handle console input.                                    */
          parse pull console_input
          say; say "Entered at the console was: '"console_input"'"
        end

      if event_name_address.index = 'VMCPIC' then
        do
          /* Handle a CPI Communications event.                       */
          data_buffer_length = 16    /* Max length for VMCPIC event */

          call CSL 'EventRetrieve retcode reascode monitor_token',
                   'index data_buffer data_buffer_length',
                   'event_data_length'

          /* The first four bytes of VMCPIC event data hold the       */
          /* event_type.                                              */
          parse var data_buffer 1 event_type +4

          /* Convert the event_type to decimal.                       */
          event_type = c2d(event_type)

          /* Determine which message format the VMCPIC event has.     */
          if event_type = xc_information_input then
            do
              /* Handle an information input event.  Format is:       */
              /* event_type||conversation_ID||event_info_length       */
              parse var data_buffer 5 conversation_ID +8,
                        event_info_length

              /* Convert the event_info_length to decimal.            */
              event_info_length = c2d(event_info_length)
              .
              .
              .
            end
          else
            do
              /* Handle an allocation request or resource revoked     */
              /* notification event.  Format is:                      */
              /* event_type||resource_ID                              */
              parse var data_buffer 5 resource_ID +8
              .
              .
              .
            end
```

```
        end
    end

end
  ⋮
```

After processing an event, the application would typically loop back to wait for the next event.

A complete description of event management services can be found in the *z/VM: CMS Application Multitasking* book.

Appendix B. CPI Communications Conversation States

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

Table 6. CPI Communications Conversation States

State	Description
Reset	There is no conversation for this *conversation_ID*.
Initialize	Initialize_Conversation has completed successfully and a *conversation_ID* has been assigned for this conversation.
Send	The program is able to send data on this conversation.
Receive	The program is able to receive data on this conversation.
Send-Pending	The program has received both data and send control on the same Receive call.
Confirm	A confirmation request has been received on this conversation; that is, the remote program issued a Confirm call and is waiting for the local program to issue Confirmed. After responding with Confirmed, the local program's end of the conversation returns to **Receive** state.
Confirm-Send	A confirmation request and send control have both been received on this conversation; that is, the remote program issued a Prepare_To_Receive call with the *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and the *sync_level* for this conversation is CM_CONFIRM. After responding with Confirmed, the local program's end of the conversation enters **Send** state.
Confirm-Deallocate	A confirmation request and deallocation notification have both been received on this conversation; that is, the remote program issued a Deallocate call with the *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL and the *sync_level* for this conversation is CM_CONFIRM. After the local program responds with Confirmed, the conversation is deallocated.

Additional CPI Communications States

In addition to the conversation states described above, the following states are required when a program uses a protected CPI Communications conversation (that is, with the *sync_level* characteristic set to CM_SYNC_POINT):

Table 7. Additional Conversation States for Protected Conversations

State	Description
Defer-Receive	The local program will enter **Receive** state after a sync point operation completes successfully; that is, the local program has issued a Prepare_to_Receive call with *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT, or it issued a Send_Data call with *send_type* set to SEND_AND_PREP_TO_RECEIVE, *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, and *sync_level* set to CM_SYNC_POINT. The conversation will not enter **Receive** state until a successful sync point operation, Flush, or Confirm takes place.

Table 7. Additional Conversation States for Protected Conversations (continued)

State	Description
Defer-Deallocate	The local program has requested to deallocate the conversation after a sync point operation has completed; that is, it issued a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT, or it issued a Send_Data call with *send_type* set to SEND_AND_DEALLOCATE, *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL, and *sync_level* set to CM_SYNC_POINT. The conversation will not be deallocated until a successful sync point operation takes place.
Sync-Point	The local program issued a Receive call and was given a *return_code* of CM_OK and a *status_received* of CM_TAKE_COMMIT. After a successful sync point operation, the conversation will return to Receive state.
Sync-Point-Send	The local program issued a Receive call and was given a *return_code* of CM_OK and a *status_received* of CM_TAKE_COMMIT_SEND. After a successful sync point operation, the conversation will be placed in **Send** state.
Sync-Point-Deallocate	The local program issued a Receive call and was given a *return_code* of CM_OK and a *status_received* of CM_TAKE_COMMIT_DEALLOCATE. After a successful sync point operation, the conversation will be deallocated and placed in **Reset** state.

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 all 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 grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, New York 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 106-0032, 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
Mail Station P300
2455 South Road
Poughkeepsie, New York 12601-5400
U.S.A.
Attention: Information Request

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 document 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 measurement 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.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information may contain 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.

COPYRIGHT LICENSE:

This information may contain sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Programming Interface Information

This book is intended to help the customer develop communications programs in VM using the Communications element of the Systems Application Architecture (SAA) Common Programming Interface (CPI) and the VM extensions to the SAA CPI Communications interface. This book documents General-Use Programming Interface and Associated Guidance Information provided by the CMS component of z/VM.

General-use programming interfaces allow the customer to write programs that obtain the services of z/VM.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml

Adobe is either a registered trademark or trademark of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, and/or other countries, or both.

Glossary

For a list of z/VM terms and their definitions, see *z/VM: Glossary*.

The z/VM glossary is also available through the online z/VM HELP Facility. For example, to display the definition of the term “dedicated device”, issue the following HELP command:

```
help glossary dedicated device
```

While you are in the glossary help file, you can do additional searches:

- To display the definition of a new term, type a new HELP command on the command line:

  ```
  help glossary newterm
  ```

  This command opens a new help file inside the previous help file. You can repeat this process many times. The status area in the lower right corner of the screen shows how many help files you have open. To close the current file, press the Quit key (PF3/F3). To exit from the HELP Facility, press the Return key (PF4/F4).
- To search for a word, phrase, or character string, type it on the command line and press the Clocate key (PF5/F5). To find other occurrences, press the key multiple times.

  The Clocate function searches from the current location to the end of the file. It does not wrap. To search the whole file, press the Top key (PF2/F2) to go to the top of the file before using Clocate.

Bibliography

See the following publications for additional information about z/VM. For abstracts of the z/VM publications, see *z/VM: General Information*.

Where to Get z/VM Information

z/VM product information is available from the following sources:

- z/VM Information Center at publib.boulder.ibm.com/infocenter/zvm/v6r1/index.jsp
- z/VM Internet Library at www.ibm.com/eserver/zseries/zvm/library/
- IBM Publications Center at www.elink.ibmlink.ibm.com/publications/servlet/pbi.wss
- *IBM Online Library: z/VM Collection on DVD*, SK5T-7054

z/VM Base Library

Overview

- *z/VM: General Information*, GC24-6193
- *z/VM: Glossary*, GC24-6195
- *z/VM: License Information*, GC24-6200

Installation, Migration, and Service

- *z/VM: Guide for Automated Installation and Service*, GC24-6197
- *z/VM: Migration Guide*, GC24-6201
- *z/VM: Service Guide*, GC24-6232
- *z/VM: VMSES/E Introduction and Reference*, GC24-6243

Planning and Administration

- *z/VM: CMS File Pool Planning, Administration, and Operation*, SC24-6167
- *z/VM: CMS Planning and Administration*, SC24-6171
- *z/VM: Connectivity*, SC24-6174
- *z/VM: CP Planning and Administration*, SC24-6178
- *z/VM: Getting Started with Linux on System z*, SC24-6194
- *z/VM: Group Control System*, SC24-6196
- *z/VM: I/O Configuration*, SC24-6198
- *z/VM: Running Guest Operating Systems*, SC24-6228
- *z/VM: Saved Segments Planning and Administration*, SC24-6229
- *z/VM: Secure Configuration Guide*, SC24-6230
- *z/VM: TCP/IP LDAP Administration Guide*, SC24-6236
- *z/VM: TCP/IP Planning and Customization*, SC24-6238
- *z/OS and z/VM: Hardware Configuration Manager User's Guide*, SC33-7989

Customization and Tuning

- *z/VM: CP Exit Customization*, SC24-6176
- *z/VM: Performance*, SC24-6208

Operation and Use

- *z/VM: CMS Commands and Utilities Reference*, SC24-6166
- *z/VM: CMS Pipelines Reference*, SC24-6169
- *z/VM: CMS Pipelines User's Guide*, SC24-6170
- *z/VM: CMS Primer*, SC24-6172
- *z/VM: CMS User's Guide*, SC24-6173
- *z/VM: CP Commands and Utilities Reference*, SC24-6175
- *z/VM: System Operation*, SC24-6233
- *z/VM: TCP/IP User's Guide*, SC24-6240
- *z/VM: Virtual Machine Operation*, SC24-6241
- *z/VM: XEDIT Commands and Macros Reference*, SC24-6244
- *z/VM: XEDIT User's Guide*, SC24-6245
- *CMS/TSO Pipelines: Author's Edition*, SL26-0018

Application Programming

- *z/VM: CMS Application Development Guide*, SC24-6162
- *z/VM: CMS Application Development Guide for Assembler*, SC24-6163
- *z/VM: CMS Application Multitasking*, SC24-6164
- *z/VM: CMS Callable Services Reference*, SC24-6165
- *z/VM: CMS Macros and Functions Reference*, SC24-6168

- *z/VM: CP Programming Services*, SC24-6179
- *z/VM: CPI Communications User's Guide*, SC24-6180
- *z/VM: Enterprise Systems Architecture/ Extended Configuration Principles of Operation*, SC24-6192
- *z/VM: Language Environment User's Guide*, SC24-6199
- *z/VM: OpenExtensions Advanced Application Programming Tools*, SC24-6202
- *z/VM: OpenExtensions Callable Services Reference*, SC24-6203
- *z/VM: OpenExtensions Commands Reference*, SC24-6204
- *z/VM: OpenExtensions POSIX Conformance Document*, GC24-6205
- *z/VM: OpenExtensions User's Guide*, SC24-6206
- *z/VM: Program Management Binder for CMS*, SC24-6211
- *z/VM: Reusable Server Kernel Programmer's Guide and Reference*, SC24-6220
- *z/VM: REXX/VM Reference*, SC24-6221
- *z/VM: REXX/VM User's Guide*, SC24-6222
- *z/VM: Systems Management Application Programming*, SC24-6234
- *z/VM: TCP/IP Programmer's Reference*, SC24-6239
- *Common Programming Interface Communications Reference*, SC26-4399
- *Common Programming Interface Resource Recovery Reference*, SC31-6821
- *z/OS: IBM Tivoli Directory Server Plug-in Reference for z/OS*, SA76-0148
- *z/OS: Language Environment Concepts Guide*, SA22-7567
- *z/OS: Language Environment Debugging Guide*, GA22-7560
- *z/OS: Language Environment Programming Guide*, SA22-7561
- *z/OS: Language Environment Programming Reference*, SA22-7562
- *z/OS: Language Environment Run-Time Messages*, SA22-7566
- *z/OS: Language Environment Writing ILC Applications*, SA22-7563
- *z/OS MVS Program Management: Advanced Facilities*, SA22-7644
- *z/OS MVS Program Management: User's Guide and Reference*, SA22-7643

Diagnosis

- *z/VM: CMS and REXX/VM Messages and Codes*, GC24-6161
- *z/VM: CP Messages and Codes*, GC24-6177
- *z/VM: Diagnosis Guide*, GC24-6187
- *z/VM: Dump Viewing Facility*, GC24-6191
- *z/VM: Other Components Messages and Codes*, GC24-6207
- *z/VM: TCP/IP Diagnosis Guide*, GC24-6235
- *z/VM: TCP/IP Messages and Codes*, GC24-6237
- *z/VM: VM Dump Tool*, GC24-6242
- *z/OS and z/VM: Hardware Configuration Definition Messages*, SC33-7986

z/VM Facilities and Features

Data Facility Storage Management Subsystem for VM

- *z/VM: DFSMS/VM Customization*, SC24-6181
- *z/VM: DFSMS/VM Diagnosis Guide*, GC24-6182
- *z/VM: DFSMS/VM Messages and Codes*, GC24-6183
- *z/VM: DFSMS/VM Planning Guide*, SC24-6184
- *z/VM: DFSMS/VM Removable Media Services*, SC24-6185
- *z/VM: DFSMS/VM Storage Administration*, SC24-6186

Directory Maintenance Facility for z/VM

- *z/VM: Directory Maintenance Facility Commands Reference*, SC24-6188
- *z/VM: Directory Maintenance Facility Messages*, GC24-6189
- *z/VM: Directory Maintenance Facility Tailoring and Administration Guide*, SC24-6190

Open Systems Adapter/Support Facility

- *System z10, System z9 and eServer zSeries: Open Systems Adapter-Express Customer's Guide and Reference*, SA22-7935
- *System z9 and eServer zSeries 890 and 990: Open Systems Adapter-Express Integrated Console Controller User's Guide*, SA22-7990

- *System z: Open Systems Adapter-Express Integrated Console Controller 3215 Support*, SA23-2247

Performance Toolkit for VM™

- *z/VM: Performance Toolkit Guide*, SC24-6209
- *z/VM: Performance Toolkit Reference*, SC24-6210

RACF® Security Server for z/VM

- *z/VM: RACF Security Server Auditor's Guide*, SC24-6212
- *z/VM: RACF Security Server Command Language Reference*, SC24-6213
- *z/VM: RACF Security Server Diagnosis Guide*, GC24-6214
- *z/VM: RACF Security Server General User's Guide*, SC24-6215
- *z/VM: RACF Security Server Macros and Interfaces*, SC24-6216
- *z/VM: RACF Security Server Messages and Codes*, GC24-6217
- *z/VM: RACF Security Server Security Administrator's Guide*, SC24-6218
- *z/VM: RACF Security Server System Programmer's Guide*, SC24-6219
- *z/VM: Security Server RACROUTE Macro Reference*, SC24-6231

Remote Spooling Communications Subsystem Networking for z/VM

- *z/VM: RSCS Networking Diagnosis*, GC24-6223
- *z/VM: RSCS Networking Exit Customization*, SC24-6224
- *z/VM: RSCS Networking Messages and Codes*, GC24-6225
- *z/VM: RSCS Networking Operation and Use*, SC24-6226
- *z/VM: RSCS Networking Planning and Configuration*, SC24-6227
- *Network Job Entry: Formats and Protocols*, SA22-7539

Prerequisite Products

Device Support Facilities

- *Device Support Facilities: User's Guide and Reference*, GC35-0033

Environmental Record Editing and Printing Program

- *Environmental Record Editing and Printing Program (EREP): Reference*, GC35-0152
- *Environmental Record Editing and Printing Program (EREP): User's Guide*, GC35-0151

Index

Special characters

$SERVER$ NAMES file 31, 168

A

accept incoming conversation request 33
Accept_Conversation (CMACCP)
  call description 33
  example flow using 60
  in sample server program 34
access security information
  setting and extracting 131
ADDRESS CPICOMM 10
advanced function calls
  description 6
  list 63
  sample program pseudocode 64
Advanced Program-to-Program Communications (APPC)
  See also LU 6.2 and CPI Communications
  interface for 1
  type 6.2 logical unit 131
Allocate (CMALLC)
  call description 17
  example flow using 60
  in sample requester program 18
allocation request 217
alter conversation characteristics
  See set calls
alternate user ID
  setting 2, 183, 191
APILOAD command 218
APPC
  See Advanced Program-to-Program Communications (APPC)
APPCPASS statement 158

B

basic conversation 6, 109
binding file 218
buffer
  description 18, 70
  example flow 60

C

calls
  advanced function 63
  format 11
  naming conventions 2
  starter set 6, 9
  VM extensions 132
change conversation characteristics
  See set calls
change conversation state
  from Send to Receive 70
change data flow direction
  by receiving program 113, 119
  by sending program 41, 46
characteristic of
  CMS files 49
  conversation_ID 12, 34
  conversation_state
    extract 65
    possible values 66
  conversation_type
    extract 105
    possible values 6
    set 109
  deallocate_type
    possible values 98
    set 98
  default values 11
  error_direction 120
  fill 120
  integer values 3
  log_data 120
  mode_name
    extract 119
    set 120
  modifying 63
  naming conventions 3
  overview 11
  partner_LU_name 119
  prepare_to_receive_type 88
  pseudonyms 3
  receive_type 121
  return_control 121
  send_type 93
  sync_level
    extract 119
    possible values 75
    set 74
  TP_name 116
  viewing 64, 132
class B privilege
  issuing DIAGNOSE code X'D4' 183, 191
  setting alternate user ID 2
client program 167
CMACCP (Accept_Conversation)
  call description 33
  example flow using 60
  in sample server program 34
CMALLC (Allocate)
  call description 17
  example flow using 60
  in sample requester program 18
CMCFM (Confirm)
  call description 80
  in sample programs 82
CMCFMD (Confirmed)
  call description 81
  in sample programs 82

CMDEAL (Deallocate)
  call description 50
  example flow using 60
  in sample requester program 50
CMECS (Extract_Conversation_State)
  call description 65
  in sample programs 66
CMECT (Extract_Conversation_Type)
  call description 105
  in sample server program 106
CMEMN (Extract_Mode_Name) 119
CMEPLN (Extract_Partner_LU_Name) 119
CMESL (Extract_Sync_Level) 119
CMFLUS (Flush) 70
CMINIT (Initialize_Conversation)
  call description 11
  example flow using 60
CMPTR (Prepare_To_Receive)
  call description 70
  in sample requester program 71
CMRCV (Receive)
  call description 36
  example flow using 60
  loop in sample requester program 41
  loop in sample server program 38, 46
CMREXX COPY file 15
CMRTS (Request_To_Send) 119
CMS multitasking
  using event management services 217
CMSCT (Set_Conversation_Type)
  call description 109
  in sample requester program 110
CMSDT (Set_Deallocate_Type)
  call description 98
  in sample programs 99
CMSED (Set_Error_Direction) 120
CMSEND (Send_Data)
  call description 20
  example flow using 60
  in sample requester program 23
  loop in sample server program 46
CMSERR (Send_Error)
  call description 107
  in sample server program 108
CMSF (Set_Fill) 64, 120
CMSLD (Set_Log_Data) 64, 120
CMSMN (Set_Mode_Name) 120
CMSPLN (Set_Partner_LU_Name)
  call description 113
  in sample requester program 114
CMSPTR (Set_Prepare_To_Receive_Type
  call description 88
  in sample requester program 89
CMSRC (Set_Return_Control) 121
CMSRT (Set_Receive_Type) 121
CMSSL (Set_Sync_Level)
  call description 74
  in sample requester program 75
CMSST (Set_Send_Type)
  call description 93
  in sample server program 94
CMSTPN (Set_TP_Name)
  call description 116
  in sample requester program 117
CMTRTS (Test_Request_To_Send_Received) 122
Common Programming Interface (CPI) Communications
  introduction 1
  naming conventions 2
communications directory
  CMS NAMES file 25
  description 11, 25
  SCOMDIR file 25
  SET COMDIR Command 27
  UCOMDIR file 25, 32
Confirm (CMCFM)
  call description 80
  in sample programs 82
Confirm state 223
Confirm-Deallocate state 223
Confirm-Send state 79, 223
confirmation processing
  Confirm call 80
  Confirmed call 81
  general discussion 77
Confirmed (CMCFMD)
  call description 81
  in sample programs 82
conventions in this book 2
conversation
  accept 33
  allocate 17
  basic 6
  characteristics 11
  deallocate 50
  description 4
  initialize 11
  mapped 6
  multiple 131
  security 156
  states 78, 223
  synchronization and control
    Confirm call 80
    Confirmed call 81
    Flush call 70
    list of calls 63
    Prepare_To_Receive call 70
    Request_To_Send call 119
    Send_Error call 107
    Test_Request_To_Send_Received call 122
  transition from a state 78
  types 6
conversation_type characteristic
  extract 105
  possible values 109
  set 109
copy file
  CMREXX 15
CPI Communications
  See Common Programming Interface (CPI) Communications
CPICOMM LOGDATA file 12
create $SERVER$ NAMES file 31

D

data
  buffering 41, 70
  direction, changing
    by receiving program 113, 119
    by sending program 41, 46
  purging 113
  transmission 41, 70
data record
  description 20, 41
  Receive call 36
  Send_Data call 20
Deallocate (CMDEAL)
  call description 50
  example flow using 60
  in sample requester program 50
deallocate_type characteristic
  possible values 98
  set 98
declare
  resource to manage 135
detect error
  using Send_Error (CMSERR) call 107
DIAGNOSE code X'D4' 183, 191

E

end conversation
  using Deallocate (CMDEAL) call 50
error
  checking 10
  detecting 107
  reporting 19, 22
error_direction characteristic 120
establish conversation
  using Allocate (CMALLC) call 17
event
  communications 217
  services 217
  system 217
event management
  using for CPI Communications 217
event management services 217
event notification 143
EventMonitorCreate
  use of 218
EventRetrieve
  use of 219
EventWait
  use of 219
examine
  See also extract calls
  conversation characteristics 64
EXECIO command 15
extension, VM
  calls
    conversation security 132, 156, 183
    overview 131, 198
    resource recovery 132
    resources, events 132
extension, VM *(continued)*
  sample program pseudocode 134
extract calls
  conversation_security_user_ID 184
  conversation_state 65
  conversation_type 105
  list of
    advanced set 64
    VM extensions 132
  mode_name 119
  partner_LU_name 119
  sync_level 119
Extract Local Fully Qualified LU Name (XCELFQ) 198
Extract Remote Fully Qualified LU Name (XCERFQ) 198
Extract_Conversation_LUWID (XCECL) 198
Extract_Conversation_Security_User_ID (XCECSU)
  call description 184
  in sample intermediate server 184
  in sample resource manager program 186
Extract_Conversation_State (CMECS)
  call description 65
  in sample programs 66
Extract_Conversation_Type (CMECT)
  call description 105
  in sample server program 106
Extract_Conversation_Workunit_ID (XCECWU) 198
Extract_Mode_Name (CMEMN) 119
Extract_Partner_LU_Name (CMEPLN) 119
Extract_Sync_Level (CMESL) 119
Extract_TP_Name (XCETPN) 198

F

file
  binding 218
  characteristics 49
fill characteristic 120
flow
  conversation 67
  diagram 60
Flush (CMFLUS) 70
flush send buffer 70
force a conversation flow using Flush (CMFLUS) 70
format of calls 11
FYI (for your information) boxes
  CMS communications directories 25
  copy files—the easy way to use pseudonyms 15
  Flush (CMFLUS) call overview 70
  if SERVR received a product-specific error 193
  if you got a product-specific error 19
  LEFT function in REXX 23, 39
  more REXX considerations 13
  receiving partial records 58
  REXX considerations 10
  security information and the APPCPASS statement 158
  SET COMDIR command 27
  side information 11
  tidying up 65
  tidying up, part II 88

FYI (for your information) boxes *(continued)*
tidying up, part III 135
what the Allocate call does 17
when errors are reported 22

G

General-Use programming interfaces 227
global resource
description 133
in communications programming 4
specifying on Identify_Resource_Manager call 136

H

half-duplex protocol 5

I

Identify_Resource_Manager (XCIDRM)
call description 135
in sample server program 137
incoming conversation request
accepting 33
information input 217
initialize
conversation 11
state 12, 223
Initialize_Conversation (CMINIT)
call description 11
example flow using 60
interface, communications
See Common Programming Interface (CPI) Communications
intermediate server
converting SERVR virtual machine 170
description 167
sample program pseudocode 167
security considerations 183
interrupt
console
reflecting 143, 217

L

local partner 4
local resource
description 133
in communications programming 4
specifying on Identify_Resource_Manager call 136
logical unit 17
LU
See logical unit
LU 6.2 and CPI Communications 131

M

manage
events 218
manager, resource 133
mapped conversation 6
mode_name characteristic
extract 119
set 120
modify conversation characteristics
See set calls
modify data flow direction
by receiving program 41, 46
by sending program 113, 119
multiple conversations 131
multitasking, CMS
using event management services 217

N

NAMES command 25, 31
NAMES file 11
naming conventions 2, 11

O

ownership of a resource
terminating 139

P

partner 4
partner_LU_name characteristic
extract 119
set 113
password
access security
setting 162
prepare
SERVR virtual machine 30
Prepare_To_Receive (CMPTR)
call description 70
in sample requester program 71
prepare_to_receive_type characteristic
possible values 89
set 88
private resource
description 134
in communications programming 4
specifying on Identify_Resource_Manager call 135
privilege class B
issuing DIAGNOSE code X'D4' 183, 191
setting alternate user ID 2
Procedures Language REXX/VM
See REstructured eXtended eXecutor/Virtual Machine (REXX/VM)
PROFILE EXEC
modifying 30
program
partners 4
states, conversation 223
programming interfaces, General-Use 227
pseudonym
copy files 15
example of 15
explanation of 3, 15

pseudonym *(continued)*
values 15

Q

query
See also extract calls
conversation characteristics 64

R

RC special variable
REXX 10
Receive (CMRCV)
call description 36
example flow using 60
loop in sample requester program 41
loop in sample server program 38, 46
receive information
using Receive (CMRCV) call 36
Receive state
description 223
how a program enters it 41, 79
receive_type characteristic
possible values 121
set 121
record
format 49
length 41, 49
partial 58
reflect
console interrupts 143, 217
related publications xi
remote partner 4
report
errors 19, 22
report events 217
Request_To_Send (CMRTS) 119
requester program 5
requester virtual machine 5
Reset state 12, 223
resource
kinds of
global 4, 133, 136
local 4, 133, 136
private 4, 134, 135
system 4, 134, 136
management 132, 133
resource revoked notification 218
REstructured eXtended eXecutor/Virtual Machine (REXX/VM)
binding files 218
considerations 10, 13
CPICOMM subcommand environment 10
functions 13
interpret statement 15
SIGNAL ON ERROR instruction 10
special variable RC 10
return
access security user ID 184
CMS work unit ID 198
return *(continued)*
conversation state 65
conversation type 105
local fully-qualified LU name 198
logical unit of work ID 198
mode name 119
partner LU name 119
remote fully-qualified LU name 198
sync_level value 119
TP name 198
return codes 3
return_control characteristic 121
REXX/VM
See REstructured eXtended eXecutor/Virtual Machine (REXX/VM)

S

SAA
See Systems Application Architecture (SAA)
sample of
completed execs 199
program pseudocode
advanced set 64
intermediate server 167
starter set 9
VM extensions 134
SCOMDIR NAMES file
See communications directory
security
$SERVER$ NAMES file 31, 183
general considerations 156
levels in VM
SECURITY(NONE) 156
SECURITY(PROGRAM) 157
SECURITY(SAME) 157
relating to intermediate servers 183
send
confirmation request 80
send data
using Send_Data (CMSEND) call 20
Send state 223
Send_Data (CMSEND)
call description 20
example flow using 60
in sample requester program 23
loop in sample server program 46
Send_Error (CMSERR)
call description 107
in sample server program 108
send_type characteristic
possible values 94
set 93
Send-Pending state 223
server program
description 4
SERVR's SENDBACK EXEC 32
SERVR2's SENDSERV EXEC 168
SERVR virtual machine, converting to intermediate server 170
SERVR2 virtual machine, setting up 168

session
description 17
set calls
client_security_user_ID 191
conversation_security_password 162
conversation_security_type 158
conversation_security_user_ID 161
conversation_type 109
deallocate_type 98
error_direction 120
fill 120
listing of 63
log_data 120
mode_name 120
partner_LU_name 113
prepare_to_receive_type 88
receive_type 121
return_control 121
send_type 93
sync_level 74
TP_name 116
SET COMDIR Command 27
set up
user IDs 2
Set_Client_Security_User_ID (XCSCUI)
call description 191
in sample intermediate server 192
Set_Conversation_Security_Password (XCSCSP)
call description 162
in sample requester program 163
Set_Conversation_Security_Type (XCSCST)
call description 158
in sample requester program 159
Set_Conversation_Security_User_ID (XCSCSU)
call description 161
in sample requester program 162
Set_Conversation_Type (CMSCT)
call description 109
in sample requester program 110
Set_Deallocate_Type (CMSDT)
call description 98
in sample programs 99
Set_Error_Direction (CMSED) 120
Set_Fill (CMSF) 64, 120
Set_Log_Data (CMSLD) 64, 120
Set_Mode_Name (CMSMN) 120
Set_Partner_LU_Name (CMSPLN)
call description 113
in sample requester program 114
Set_Prepare_To_Receive_Type (CMSPTR)
call description 88
in sample requester program 89
Set_Receive_Type (CMSRT) 121
Set_Return_Control (CMSRC) 121
Set_Send_Type (CMSST)
call description 93
in sample server program 94
Set_Sync_Level (CMSSL)
call description 74
in sample requester program 75
Set_TP_Name (CMSTPN)
call description 116
in sample requester program 117
side information
general information 11, 25
in VM/ESA 25
Signal_User_Event (XCSUE) 198
SNA
See Systems Network Architecture (SNA)
starter-set calls
description 6
list 9
sample program call table 9
sample program flow 58
state
conversation
description 223
extracting 65
list 223
state table for conversations
abbreviations 79
example of how to use 78
subcommand environment
in REXX 10
symbolic destination name 12
sync_level characteristic
extract 119
possible values 75
set 74
synchronization and control calls
Confirm 80
Confirmed 81
Flush 70
list of 63
Prepare_To_Receive 70
Request_To_Send 119
Send_Error 107
Test_Request_To_Send_Received 122
system resource
description 134
in communications programming 4
specifying on Identify_Resource_Manager call 136
Systems Application Architecture (SAA)
See also Common Programming Interface (CPI) Communications
overview 1, 131
Systems Network Architecture (SNA) 131

T

Terminate_Resource_Manager (XCTRRM)
call description 139
in sample server program 140
Test_Request_To_Send_Received (CMTRTS) 122
TP_name characteristic 116
transition, state 79
Transparent Services Access Facility (TSAF) 19
types of conversations 6

U

UCOMDIR NAMES file
  See communications directory
user ID
  setting up 2
user program 5

V

variables
  integer values 3
  pseudonyms 3
view conversation characteristics
  See extract calls
VM extension
  calls
    conversation security 132, 156, 183
    overview 131, 198
    resource recovery 132
    resources, events 132
  sample program pseudocode 134
VM/REXX
  See REstructured eXtended eXecutor/Virtual Machine (REXX/VM)
VMCONINPUT system event 217
VMCPIC system event 217

W

Wait_on_Event (XCWOE)
  call description 143
  in sample server program 145

X

XCECL (Extract_Conversation_LUWID) 198
XCECSU (Extract_Conversation_Security_User_ID)
  call description 184
  in sample intermediate server 184
  in sample resource manager program 186
XCECWU (Extract_Conversation_Workunit_ID) 198
XCELFQ (Extract Local Fully Qualified LU Name) 198
XCERFQ (Extract Remote Fully Qualified LU Name) 198
XCETPN (Extract_TP_Name) 198
XCIDRM (Identify_Resource_Manager)
  call description 135
  in sample server program 137
XCSCSP (Set_Conversation_Security_Password)
  call description 162
  in sample requester program 163
XCSCST (Set_Conversation_Security_Type)
  call description 158
  in sample requester program 159
XCSCSU (Set_Conversation_Security_User_ID)
  call description 161
  in sample requester program 162
XCSCUI (Set_Client_Security_User_ID)
  call description 191
  in sample intermediate server 192
XCSUE (Signal_User_Event) 198
XCTRRM (Terminate_Resource_Manager)
  call description 139
  in sample server program 140
XCWOE (Wait_on_Event)
  call description 143
  in sample server program 145

Program Number: 5741-A07

Printed in USA

SC24-6180-00