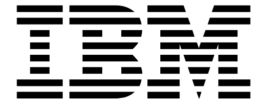
Common Programming Interface
Communications

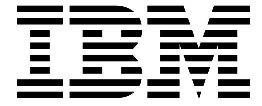


CPI-C Reference

Version 2.1

SC26-4399-09

Common Programming Interface
Communications



CPI-C Reference

Version 2.1

SC26-4399-09

Note

Before using this document, read the general information under "Notices" on page xxiii.

Tenth Edition (July 1998)

This edition replaces the previous edition, SC26-4399-08.

This edition applies to the Common Programming Interface Communications (CPI-C) architecture and to the following:

- Version 3 Release 1 of AIX SNA Server/6000 (program number 5765-582)
- Version 2 Release 1.2 of AIX SNA Server/6000 (program number 5601-287)
- Version 1 Release 1 of Desktop SNA for AIX (program number 5765-419)
- Version 3 Release 3 of CICS/ESA (program number 5685-083)
- Version 1 of IBM Communications Manager/2 (part number 20G1575 – 3½")
- Version 1 of IBM Communications Manager/2 (part number 53G3769 – 5¼")
- Version 1.1 of IBM Communications Manager/2 (part number 79G0257 – CD-ROM)
- Version 1.1 of IBM Communications Manager/2 (part number 79G0258 – 5½")
- Version 1.11 of IBM Communications Manager/2 (part number 79G0257 – CD-ROM)
- Version 1.11 of IBM Communications Manager/2 (part number 79G0258 – 5½")
- Version 4.0 of IBM Communications Server for OS/2 WARP (part number 33H7328 – CD-ROM)
- Version 4.1 of Personal Communications AS/400 and 3270 for OS/2 (part number 39H3929 – 3½")
- Version 5 of IMS/ESA Transaction Manager (program number 5695-176)
- Version 5 of MVS/ESA System Product (program numbers 5655-068 and 5655-069)
- Version 1.0 of IBM APPC Networking Services for Windows (part number 11H0400)
- Version 3 Release 1 Modification 0 of Operating System/400 (program number 5763-SS1)
- Release 2.2 of VM/ESA (program number 5684-112)
- Version 4 Release 1 of IBM eNetwork Personal Communications AS/400 and 3270 for Windows 95 (part number 69H0–342)
- Version 4 Release 1 of IBM eNetwork Personal Communications AS/400 for Windows 95 (part number 64H0–375)
- Version 5 of IBM eNetwork Communication Server for Windows NT
- Version 2 Release 2 of Netware for SAA
- Version 2 Release 3 of IntraNetware for SAA
- Version 4 Release 1 of IBM eNetwork Personal Communications for WinNT
- Version 4 Release 2 of IBM eNetwork Personal Communications for Win95

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

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

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

Publications are not stocked at the address given below. If you want more IBM publications, ask your IBM representative or write to the IBM branch office serving your locality. In the United States, you may also order IBM publications by calling 1-800-879-2755.

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

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

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

© **Copyright International Business Machines Corporation 1996, 1998. All rights reserved.**
Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Notices . . . xxiii
Programming Interface Information . . . xxiv
Trademarks and Service Marks . . . xxiv

Acknowledgements . . . xxv

Part 1. CPI-C 2.0 Architecture . . . 1

Chapter 1. Introduction . . . 5
CPI-C and the Conversational Model . . . 5
Who Should Read This Book . . . 5
What Is New in This Book . . . 6
Relationship to CPI-C 2.1 Specification . . . 6
Relationship to Products . . . 7
Additional Information Sources . . . 8
Previous Editions of this Reference . . . 8
Related APPC/LU 6.2 Publications . . . 8
CD-ROM . . . 9
Solution Developers Organization . . . 9
Online Information—CPI-C, APPC, and APPN . . . 9
CompuServe . . . 9
OS/2 BBS . . . 9
Internet . . . 10
Functional Levels of CPI Communications . . . 10
CPI-C 1.0 . . . 10
CPI-C 1.1 . . . 10
X/Open CPI-C . . . 10
CPI-C 1.2 . . . 10
CPI-C 2.0 . . . 11
X/Open CPI-C 2.0 . . . 11
CPI-C 2.1 . . . 11
Call Table for Functional Levels of CPI-C . . . 12
Naming Conventions—Calls, Characteristics, Variables, and Values . . . 13

Chapter 2. CPI Communications Terms and Concepts . . . 17
Communication across a Network . . . 18
Conversation Types . . . 19
Send-Receive Modes . . . 19
Program Partners . . . 20
Identifying the Partner Program . . . 20
Operating Environment . . . 21
Node Services . . . 22
Side Information . . . 23
Distributed Directory . . . 25
CPI Communications Directory Object . . . 25
Using the Distributed Directory . . . 26
Interaction with Side Information and Set Calls . . . 27
Distributed Security . . . 28
Operating System . . . 28

© Copyright IBM Corp. 1996, 1998

Program Calls . . . 29
Starter Set Calls . . . 29
Establishing a Conversation . . . 29
Multiple Conversations . . . 30
Partner Program Names . . . 30
Multiple Outbound Conversations . . . 30
Multiple Inbound Conversations . . . 31
Contexts and Context Management . . . 32
Relationship between Contexts and Conversations . . . 32
Relationship between Contexts and Security Parameters . . . 32
Inbound and Outbound Conversations . . . 32
Conversation Characteristics . . . 33
Modifying and Viewing Characteristics . . . 34
Characteristic Values and CRMs . . . 39
Characteristic Values and Send-Receive Modes . . . 40
Automatic Conversion of Characteristics . . . 41
Automatic Data Conversion . . . 42
Data Conversion . . . 43
Data Buffering and Transmission . . . 44
Concurrent Operations . . . 44
Using Multiple Program Threads . . . 45
Non-Blocking Operations . . . 47
Conversation-Level Non-Blocking . . . 48
Queue-Level Non-Blocking . . . 49
Working with Wait Facility . . . 49
Wait Facility Scenario . . . 49
Using Callback Function . . . 50
Canceling Outstanding Operations . . . 50
Non-Blocking Calls and Context Management . . . 50
Conversation Security . . . 51
Program Flow—States and Transitions . . . 52
Support for Resource Recovery Interfaces . . . 54
Coordination with Resource Recovery Interfaces . . . 55
Take-Commit and Take-Backout Notifications . . . 55
The Backout-Required Condition . . . 58
Responses to Take-Commit and Take-Backout Notifications . . . 59
Chained and Unchained Transactions . . . 61
Joining a Transaction . . . 61
Superior and Subordinate Programs . . . 63
Additional CPI Communications States . . . 64
Valid States for Resource Recovery Calls . . . 65
TX Extensions for CPI Communications . . . 66

Chapter 3. Program-to-Program Communication Example Flows . . . 67
Interpreting the Flow Diagrams . . . 67
Starter-Set Flows . . . 68
Example 1: Data Flow in One Direction . . . 69
Example 2: Data Flow in Both Directions . . . 72
Controlling Data Flow Direction . . . 74
Example 3: The Sending Program Changes the Data Flow Direction . . . 74
Example 4: The Receiving Program Changes the Data Flow Direction . . . 75
Verifying Receipt of Data . . . 78
Example 5: Validation of Data Receipt . . . 78
Reporting Errors to Partner . . . 80

Example 6: Reporting Errors . . . 80
Example 7: Error Direction and Send-Pending State . . . 82
Using Full-Duplex Conversations . . . 84
Example 8: Establishing a Full-Duplex Conversation . . . 84
Example 9: Using a Full-Duplex Conversation . . . 86
Example 10: Terminating a Full-Duplex Conversation . . . 88
Using Queue-Level Non-Blocking . . . 90
Example 11: Queue-Level Non-Blocking . . . 90
Accepting Multiple Conversations . . . 92
Example 12: Accepting Multiple Conversations Using Blocking Calls . . . 92
Example 13: Accepting Multiple Conversations Using Conversation-Level Non-Blocking Calls . . . 94
Using the Distributed Directory . . . 96
Example 14: Using the Distributed Directory to Find the Partner Program . . . 96
Resource Recovery Flows . . . 98
Example 15: Sending Program Issues a Commit . . . 98
Example 16: Successful Commit with Conversation State Change . . . 100
Example 17: Conversation Deallocation before the Commit Call . . . 102

Part 2. CPI-C 2.1 Call Reference . . . 105

Chapter 4. Call Reference . . . 107
Call Syntax . . . 108
Conformance Class and Interface Definition Table . . . 109
Programming Language Considerations . . . 111
Application Generator . . . 112
C . . . 112
COBOL . . . 112
FORTRAN . . . 112
PL/I . . . 112
REXX . . . 113
RPG . . . 113
How to Use the Call References . . . 113
Summary List of Calls and Their Descriptions . . . 114
Accept_Conversation (CMACCP) . . . 119
Accept_Incoming (CMACCI) . . . 121
Allocate (CMALLC) . . . 124
Cancel_Conversation (CMCANC) . . . 131
Confirm (CMCFM) . . . 133
Confirmed (CMCFMD) . . . 137
Convert_Incoming (CMCNVI) . . . 139
Convert_Outgoing (CMCNVO) . . . 141
Deallocate (CMDEAL) . . . 143
Deferred_Deallocate (CMDFDE) . . . 153
Extract_AE_Qualifier (CMEAEQ) . . . 155
Extract_AP_Title (CMEAPT) . . . 157
Extract_Application_Context_Name (CMEACN) . . . 159
Extract_Conversation_Context (CMECTX) . . . 161
Extract_Conversation_State (CMECS) . . . 163
Extract_Conversation_Type (CMECT) . . . 166
Extract_Initialization_Data (CMEID) . . . 168
Extract_Mapped_Initialization_Data (CMEMID) . . . 170
Extract_Maximum_Buffer_Size (CMEMBS) . . . 173

Extract_Mode_Name (CMEMN) 175
Extract_Partner_ID (CMEPID) 177
Extract_Partner_LU_Name (CMEPLN) 180
Extract_Secondary_Information (CMESI) 182
Extract_Security_User_ID (CMESUI) 185
Extract_Send_Receive_Mode (CMESRM) 187
Extract_Sync_Level (CMESL) 189
Extract_TP_Name (CMETPN) 191
Extract_Transaction_Control (CMETC) 193
Flush (CMFLUS) 195
Include_Partner_In_Transaction (CMINCL) 198
Initialize_Conversation (CMINIT) 200
Initialize_For_Incoming (CMINIC) 203
Prepare (CMPREP) 205
Prepare_To_Receive (CMPTR) 208
Receive (CMRCV) 213
Receive_Expedited_Data (CMRCVX) 228
Receive_Mapped_Data (CMRCVM) 231
Release_Local_TP_Name (CMRLTP) 244
Request_To_Send (CMRTS) 246
Send_Data (CMSEND) 249
Send_Error (CMSERR) 259
Send_Expedited_Data (CMSNDX) 268
Send_Mapped_Data (CMSNDM) 271
Set_AE_Qualifier (CMSAEQ) 280
Set_Allocate_Confirm (CMSAC) 282
Set_AP_Title (CMSAPT) 284
Set_Application_Context_Name (CMSACN) 286
Set_Begin_Transaction (CMSBT) 288
Set_Confirmation_Urgency (CMSCU) 290
Set_Conversation_Security_Password (CMSCSP) 292
Set_Conversation_Security_Type (CMSCST) 295
Set_Conversation_Security_User_ID (CMSCSU) 298
Set_Conversation_Type (CMSCT) 301
Set_Deallocate_Type (CMSDT) 303
Set_Error_Direction (CMSED) 307
Set_Fill (CMSF) 310
Set_Initialization_Data (CMSID) 312
Set_Join_Transaction (CMSJT) 314
Set_Log_Data (CMSLD) 316
Set_Mapped_Initialization_Data (CMSMID) 318
Set_Mode_Name (CMSMN) 321
Set_Partner_ID (CMSPID) 323
Set_Partner_LU_Name (CMSPLN) 327
Set_Prepare_Data_Permitted (CMSPDP) 329
Set_Prepare_To_Receive_Type (CMSPTR) 331
Set_Processing_Mode (CMSPM) 334
Set_Queue_Callback_Function (CMSQCF) 337
Set_Queue_Processing_Mode (CMSQPM) 340
Set_Receive_Type (CMSRT) 344
Set_Return_Control (CMSRC) 346
Set_Send_Receive_Mode (CMSSRM) 349
Set_Send_Type (CMSST) 351
Set_Sync_Level (CMSSL) 354

Set_TP_Name (CMSTPN) 357
Set_Transaction_Control (CMSTC) 359
Specify_Local_TP_Name (CMSLTP) 361
Test_Request_To_Send_Received (CMTRTS) 363
Wait_For_Completion (CMWCMP) 366
Wait_For_Conversation (CMWAIT) 369

Part 3. CPI-C 2.1 Implementation Specifics 373

Chapter 5. CPI Communications on AIX 381
AIX Publications 381
AIX Operating Environment 382
  AIX CPI Communications Concepts 382
  Conformance Classes Supported 382
  Languages Supported 383
  Pseudonym Files 383
  Profiles 383
  Creating and Maintaining Profiles through SMIT 388
  Starting SMIT 388
  Working with Profiles 388
  Verifying Profiles 388
  How Dangling Conversations Are Deallocated 388
  Scope of the Conversation_ID 389
  Identifying Product-Specific Errors 389
  Diagnosing Errors 389
  When Allocation Requests Are Sent 391
  Deviations from the CPI Communications Architecture 391
  Security Using CPI Communications and AIX 391
  Compilation 393
  Running a Transaction Program 393
AIX Extension Calls 394
Extract_Conversation_Security_Type (XCECST) 396
Extract_Conversation_Security_User_ID (XCECSU) 398
Set_Conversation_Security_Password (XCSCSP) 399
Set_Conversation_Security_Type (XCSCST) 401
Set_Conversation_Security_User_ID (XCSCSU) 403
Set_Signal_Behavior (XCSSB) 405

Chapter 6. CPI Communications on CICS/ESA 407
CICS/ESA Publications 407
CICS/ESA Operating Environment 407
  Conformance Classes Supported 408
  Languages Supported 408
  Pseudonym Files 408
  Defining Side Information 409
  How Dangling Conversations Are Deallocated 411
  Scope of the Conversation_ID 412
  Identifying Product-Specific Errors 412
  Diagnosing Errors 412
  When Allocation Requests are Sent 412
  Deviations from the CPI Communications Architecture 413
CICS/ESA Extension Calls 413
CICS/ESA Special Notes 414

Chapter 7. CPI Communications on IMS/ESA 415

Chapter 8. CPI Communications on MVS/ESA 417
MVS/ESA Publications 417
MVS/ESA Operating Environment 418
  Conformance Classes Supported 418
  Languages Supported 419
  Pseudonym Files 419
  Defining Side Information 420
  How Dangling Conversations Are Deallocated 420
  Scope of the Conversation_ID 420
  Identifying Product-Specific Errors 421
  Diagnosing Errors 423
  When Allocation Requests Are Sent 424
  Deviations from the CPI Communications Architecture 424
MVS/ESA Extension Calls 425
MVS/ESA Special Notes 425
  TP Profiles 425
  MVS Performance Considerations 425
  APPC/MVS Services 425

Chapter 9. CPI Communications on Networking Services for Windows 429
Networking Services for Windows Publications 429
Networking Services for Windows Operating Environment 429
  Support of CPI-C Conformance Classes 429
  Optional Conformance Classes Supported 429
  Optional Conformance Classes Not Supported 430
Languages Supported 430
Pseudonym Files 430
  Examples of Using C 431
    CPI-C Function Calls in C 431
    Using the Pseudonym Files in C Language Programs 431
    Using Other Languages 431
Linking with the CPI-C Import Library 432
Memory Considerations 432
  Data Buffers 432
  Stack Size 432
Defining Side Information 432
Usage Notes for Mode_Name and TP_Name 432
  Mode_Name 432
  Restrictions on Transaction Program Names 433
How Dangling Conversations Are Deallocated 433
Diagnosing Errors 433
  Log_Data 433
  Identifying Product-Specific Errors 433
Deviations from the CPI Communications Architecture 434
  Return_control Characteristic for Allocate (CMALLC) 434
  CM_PROGRAM_PARAMETER_CHECK Return Code 434
  Log Data Support 434

Chapter 10. CPI Communications on OS/2 435
OS/2 Publications 436
OS/2 Operating Environment 437
  Conformance Classes Supported 437

Languages Supported . . . 437
C . . . 438
COBOL . . . 439
FORTRAN . . . 439
REXX (SAA Procedures Language) . . . 440
Pseudonym Files . . . 443
Defining Side Information . . . 443
User-Defined Side Information . . . 444
Program-Defined Side Information . . . 444
Side Information Parameters . . . 445
How Dangling Conversations Are Deallocated . . . 447
Scope of the Conversation_ID . . . 447
Identifying Product-Specific Errors . . . 447
Diagnosing Errors . . . 448
Set_Log_Data (CMSLD) . . . 448
Logging Errors for CPI Communications Error Return Codes . . . 448
Causes for the CM_PROGRAM_PARAMETER_CHECK Return Code . 449
Causes for the CM_PROGRAM_STATE_CHECK Return Code . . . 449
When Allocation Requests Are Sent . . . 450
Deviations from the CPI Communications Architecture . . . 450
Accept_Incoming (CMACCI) . . . 450
Release_Local_TP_Name (CMRLTP) . . . 451
Specify_Local_TP_Name (CMSLTP) . . . 451
Set_Sync_Level (CMSSL) . . . 451
Programming Languages Not Supported . . . 452
Mode Names Not Supported . . . 452
CPI Communications Functions Not Available . . . 452
OS/2 Extension Calls—System Management . . . 454
Delete_CPIC_Side_Information (XCMDSI) . . . 455
Extract_CPIC_Side_Information (XCMESI) . . . 457
Set_CPIC_Side_Information (XCMSSI) . . . 460
Define_TP (XCDEFTP) . . . 463
Delete_TP (XCDELTP) . . . 467
Register_Memory_Object (XCRMO) . . . 469
Unregister_Memory_Object (XCURMO) . . . 470
OS/2 Extension Calls—Conversation . . . 471
Extract_Conversation_Security_Type (XCECST) . . . 472
Extract_Conversation_Security_User_ID (XCECSU) . . . 473
Initialize_Conv_For_TP (XCINCT) . . . 474
Set_Conversation_Security_Password (XCSCSP) . . . 476
Set_Conversation_Security_Type (XCSCST) . . . 477
Set_Conversation_Security_User_ID (XCSCSU) . . . 478
OS/2 Extension Calls—Transaction Program Control . . . 479
End_TP (XCENDT) . . . 480
Extract_TP_ID (XCETI) . . . 482
Start_TP (XCSTP) . . . 483
OS/2 Special Notes . . . 485
Migration to Communications Server . . . 485
Multi-threaded CPI-C Programs . . . 485
Considerations for CPI Communications Calls . . . 485
TP Instances for Communications Manager . . . 486
Accept_Conversation (CMACCP) or Accept_Incoming (CMACCI) . . . 487
Extract_Conversation_Context (CMECTX) . . . 488
Extract_Secondary_Information (CMESI) . . . 489

Initialize_Conversation (CMINIT) 490
Receive (CMRCV) 492
Send_Data (CMSEND) 492
Send_Expedited_Data (CMSNDX) 493
Set_Partner_LU_Name (CMSPLN) 493
Set_Sync_Level (CMSSL) 493
Set_Queue_Processing_Mode (CMSQPM) 493
Test_Request_To_Send (CMTRTS) 493
Wait_For_Completion (CMWCMP) 493
Characteristics, Fields, and Variables 494
Communications Manager Native Encoding 494
Variable Types and Lengths 495
Defining and Running a CPI Communications Program on Communications Manager 498
Defining a CPI Communications Program to Communications Manager 498
Using Defaults for TP Definitions 498
Communications Manager Use of OS/2 Environment Variables 499
Stack Size 500
Performance Considerations For Using Send/Receive Buffers 500
Exit List Processing 501
Sample Program Listings for OS/2 502
OS/2 C Sample Programs 503
SETSIDE.C 503
OS/2 COBOL Sample Programs 504
DEFSIDE.CBL 504
DELSIDE.CBL 507
OS/2 REXX Sample Programs 509
XCMSSI.CMD 509
XCMESI.CMD 510

Chapter 11. CPI Communications on Operating System/400 513
OS/400 Publications 513
OS/400 Operating Environment 513
OS/400 Terms and Concepts 513
Conformance Classes Supported 515
Languages Supported 515
Pseudonym Files 516
Defining Side Information 516
Managing the Communications Side Information 517
How Dangling Conversations Are Deallocated 518
Reclaim Resource Processing 519
Scope of the Conversation_ID 519
Identifying Product-Specific Errors 519
CM_PRODUCT_SPECIFIC_ERROR 519
Diagnosing Errors 520
OS/400 CPI Communications Support of Log_Data 521
Return Codes 521
REXX Reserved RC Variable 522
REXX Error and Failure Conditions 523
Tracing CPI Communications 523
When Allocation Requests Are Sent 524
OS/400 Extension Calls 524
OS/400 Special Notes 524
CPI Communications over TCP/IP Support 524

Prestarting Jobs for Incoming Conversations 524
Multiple Conversation Support 525
Portability Considerations 525

Chapter 12. CPI Communications on VM/ESA CMS 527
VM Publications 527
VM/ESA Operating Environment 528
Conformance Classes Supported 528
Languages Supported 528
Programming Language Considerations 529
Pseudonym Files 530
Defining Side Information 532
How Dangling Conversations Are Deallocated 534
Scope of the Conversation_ID 534
Identifying Product-Specific Errors 534
Diagnosing Errors 536
Processing Log Data 536
Invocation Errors 537
Possible Causes for Selected Return Codes 538
APPC Protocol Errors in VM/ESA 539
When Allocation Requests Are Sent 540
Deviations from the CPI Communications Architecture 540
VM/ESA Extension Calls 541
Extract_Conversation_LUWID (XCECL) 544
Extract_Conversation_Security_User_ID (XCECSU) 546
Extract_Conversation_Workunitid (XCECWU) 548
Extract_Local_Fully_Qualified_LU_Name (XCELFQ) 550
Extract_Remote_Fully_Qualified_LU_Name (XCERFQ) 552
Extract_TP_Name (XCETPN) 554
Identify_Resource_Manager (XCIDRM) 555
Set_Client_Security_User_ID (XCSCUI) 559
Set_Conversation_Security_Password (XCSCSP) 562
Set_Conversation_Security_Type (XCSCST) 564
Set_Conversation_Security_User_ID (XCSCSU) 566
Signal_User_Event (XCSUE) 568
Terminate_Resource_Manager (XCTRRM) 570
Wait_on_Event (XCWOE) 571
VM/ESA Variables and Characteristics 576
Pseudonyms and Integer Values 576
Variable Types and Lengths 577
VM/ESA Special Notes 577
Program-Startup Processing 578
End-of-Command Processing 578
Work Units 578
External Interrupts 579
Coordination with the SAA Resource Recovery Interface 579
Additional Conversation Characteristics 579
TP-Model Applications in VM/ESA 580
LU 6.2 Communications Model 580
VM/ESA TP-Model Applications 581
Implications 582
VM/ESA-Specific Notes for CPI Communications Routines 583
VM/ESA Communications Events 585
The VMCPIC Event 586

Notes on the VMCPIC Event . . . 587
Using the Online HELP Facility . . . 588

Chapter 13. CPI Communications on IBM eNetwork Personal Communications V4.1 for Windows 95 . . . 589
Conformance Classes Supported . . . 589
Personal Communications V4.1 for Windows 95 Publications . . . 590
Programming Language Support . . . 590
Linking with the CPI-C library . . . 590
Accepting Conversations . . . 590
Extension Calls supported . . . 591
WinCPICStartup . . . 591
WinCPICCleanup . . . 591
Specify_Windows_Handle (XCHWND) . . . 591
Deviations from the CPI-C architecture . . . 592

Chapter 14. CPI Communications on Win32 and 32-bit API Client Platforms . . . 593
Operating Environment . . . 594
Conformance Classes Supported . . . 594
Languages Supported . . . 596
CPI-C Communications Use of Environment Variables . . . 597
Pseudonym Files . . . 597
Defining Side Information . . . 598
User-Defined Side Information . . . 598
Program-Defined Side Information . . . 598
How Dangling Conversations Are Deallocated . . . 599
Diagnosing Errors . . . 599
Causes for the CM_PROGRAM_PARAMETER_CHECK Return Code . 599
Causes for the CM_PROGRAM_STATE_CHECK Return Code . . . 599
When Allocation Requests Are Sent . . . 600
Deviations from the CPI Communications Architecture . . . 600
Accept_Incoming (CMACCI) . . . 600
Release_Local_TP_Name (CMRLTP) . . . 600
Mode Names Not Supported . . . 600
CPI-C Communication Functions Not Available . . . 600
Extension Calls – System Management . . . 601
Delete_CPIC_Side_Information (XCMDSI) . . . 602
Extract_CPIC_Side_Information . . . 604
Set_CPIC_Side_Information . . . 607
Extension Calls—Conversation . . . 610
Extract_Conversation_Security_Type (XCECST) . . . 611
Extract_Conversation_Security_User_ID (XCECSU) . . . 612
Initialize_Conv_For_TP (XCINCT) . . . 613
Set_Conversation_Security_Password (XCSCSP) . . . 615
Set_Conversation_Security_Type (XCSCST) . . . 616
Set_Conversation_Security_User_ID (XCSCSU) . . . 617
Extension Calls—Transaction Program Control . . . 618
End_TP (XCENDT) . . . 619
Extract_TP_ID (XCETI) . . . 621
Start_TP (XCSTP) . . . 622
Special Notes . . . 624
Migration to Communications Server . . . 624
Multi-threaded CPI-C Programs . . . 624

Considerations for CPI Communications Calls . . . 624
TP Instances for CPI-C Communications . . . 625
Accept_Conversation (CMACCP) or Accept_Incoming (CMACCI) . . . 625
Extract_Conversation_Context (CMECTX) . . . 627
Extract_Secondary_Information (CMESI) . . . 627
Initialize_Conversation (CMINIT) . . . 628
Receive (CMRCV) . . . 629
Send_Data (CMSEND) . . . 630
Send_Expedited_Data (CMSNDX) . . . 630
Set_Partner_LU_Name (CMSPLN) . . . 630
Set_Queue_Processing_Mode (CMSQPM) . . . 630
Test_Request_To_Send (CMTRTS) . . . 631
Wait_For_Completion (CMWCMP) . . . 631
Characteristics, Fields, and Variables . . . 631
Variable Types and Lengths . . . 632
WOSA Extension Calls Supported . . . 633
WinCPICStartup . . . 634
Returns . . . 634
WinCPICCleanup . . . 634
WINCPICCleanup() . . . 634
Specify_Windows Handle (xchwnd) . . . 635

Part 4. CPI-C 2.1 Appendixes . . . 637

Appendix A. Variables and Characteristics . . . 641
Pseudonyms and Integer Values . . . 641
Character Sets . . . 647
Variable Types . . . 649
Integers . . . 649
Character Strings . . . 649
Distinguished Name . . . 655
Program Function Identifier (PFID) . . . 656
PFID Assignment Algorithms . . . 656
Program Binding . . . 658

Appendix B. Return Codes and Secondary Information . . . 661
Return Codes . . . 661
Secondary Information . . . 679
Application-Oriented Information . . . 680
CPI Communications-Defined Information . . . 681
CRM-Specific Secondary Information . . . 692
Implementation-Related Information . . . 693

Appendix C. State Tables . . . 695
How to Use the State Tables . . . 695
Example . . . 696
Explanation of Half-Duplex State Table Abbreviations . . . 697
Conversation Characteristics ( ) . . . 697
Conversation Queues () . . . 699
Return Code Values [ ] . . . 700
data_received and status_received { , } . . . 702
Table Symbols for the Half-Duplex State Table . . . 703

Effects of Calls to the SAA Resource Recovery Interface on Half-Duplex Conversations . . . 710
Effects of Calls on Half-Duplex Conversations to the X/Open TX Interface . 711
Explanation of Full-Duplex State Table Abbreviations . . . 712
  Conversation Characteristics ( ) . . . 712
  Conversation Queues () . . . 713
  Return Code Values [ ] . . . 713
  data_received and status_received { , } . . . 715
  Table Symbols for the Full-Duplex State Table . . . 716
Effects of Calls to the SAA Resource Recovery Interface on Full-Duplex Conversations . . . 723
Effects of Calls on Full-Duplex Conversations to the X/Open TX Interface . . 724

Appendix D. CPI Communications and LU 6.2 . . . 725
  Send-Pending State and the error_direction Characteristic . . . 726
  Can CPI Communications Programs Communicate with APPC Programs? 727
  SNA Service Transaction Programs . . . 727
  Implementation Considerations . . . 727
  Relationship between LU 6.2 Verbs and CPI Communications Calls . . . 727

Appendix E. Application Migration from X/Open CPI-C . . . 735

Appendix F. CPI Communications Extensions for Use with DCE Directory . . . 737
Profile Object . . . 737
Server Object . . . 737
Server Group Object . . . 738
Interaction of Directory Objects . . . 738
CPI-C Name Service Interface . . . 739
  CNSI Calls . . . 740
Definition of New Objects . . . 740
  Terminology . . . 740
  Profile Object . . . 741
  Server Object . . . 741
  Server Group Object . . . 741
  Program Installation Object . . . 742
  Encoding Method for Complex Attribute Values . . . 742
Scenarios for Use of CNSI . . . 742
  (PFID, *, *) . . . 743
  (PFID, SDN, *) . . . 743
  (PFID, SGDN, *) . . . 743
  (PFID, SDN, resID) . . . 744
  (PFID, SGDN, resID) . . . 744
  (PFID, PDN, *) . . . 744
  (PFID, PDN, resID) . . . 744

Appendix G. CPI Communications 2.1 Conformance Classes . . . 745
Definitions . . . 745
Conformance Requirements . . . 745
  Multi-Threading Support . . . 745
CPI-C 2.1 Conformance Classes . . . 745
Functional Conformance Class Descriptions . . . 746
  Conversations . . . 746
  LU 6.2 . . . 747

OSI TP . . . 747
Recoverable Transactions . . . 748
Unchained Transactions . . . 748
Conversation-Level Non-Blocking . . . 749
Queue-Level Non-Blocking . . . 749
Callback Function . . . 749
Server . . . 750
Data Conversion Routines . . . 750
Security . . . 750
Distributed Security . . . 751
Full-Duplex . . . 751
Expedited Data . . . 751
Directory . . . 751
Secondary Information . . . 752
Initialization Data . . . 752
Automatic Data Conversation . . . 752
Configuration Conformance Class Description . . . 753
OSI TP Addressing Disable . . . 753
Relationship to OSI TP Functional Units and OSI TP Profiles . . . 754
Conformance Class Details . . . 755

Appendix H. Solution Developers Program - Enterprise Communications Partners in Development . . . 763
Program Highlights . . . 763
Membership . . . 763

Glossary . . . 765

Index . . . 769

Figures

1. Programs Using CPI Communications to Converse through a Network . 18
2. Operating Environment of CPI Communications Program 22
3. Generic Program Interaction with a Distributed Directory 25
4. Program Interaction with CPI Communications and a Distributed Directory 27
5. CRM Interaction with Distributed Security Service 28
6. A Program Using Multiple Outbound CPI Communications Conversations 31
7. A Program Using Multiple Inbound CPI Communications Conversations 31
8. Server Program with Both Inbound and Outbound Conversations 33
9. Commit Tree with Program 1 as Root and Superior 63
10. Data Flow in One Direction 71
11. Data Flow in Both Directions 73
12. The Sending Program Changes the Data Flow Direction 75
13. Changing the Data Flow Direction 77
14. Validation and Confirmation of Data Reception 79
15. Reporting Errors 81
16. Error Direction and Send-Pending State 83
17. Establishing a Full-Duplex Conversation 85
18. Using a Full-Duplex Conversation 87
19. Terminating a Full-Duplex Conversation 89
20. Using Queue-Level Non-Blocking 91
21. Accepting Multiple Conversations Using Blocking Calls 93
22. Accepting Multiple Conversations Using Non-Blocking Calls 95
23. Using the Distributed Directory to Locate the Partner Program 97
24. Establishing a Protected Conversation and Issuing a Successful Commit 99
25. A Successful Commit with Conversation State Change 101
26. Conversation Deallocation Precedes the Commit Call 103
27. LU 6.2 Communications Model 580
28. Creating a TP-Model Application in VM/ESA 581
29. Three Potential Conversation Wrap-Back Scenarios 583
30. Relationship of PFID to Program Installation DNs 656
31. Sample Program Binding Format 659
32. Interactions of Directory Objects 738
33. CPI-C Name Service Interface (CNSI) and CPI Communications 739
34. Program Uses CNSI to Locate PIDN 743

© Copyright IBM Corp. 1996, 1998

Tables

1. Previous Editions of the CPI-C Reference 8
2. Versions of CPI Communications 12
3. Characteristics and Their Default Values 35
4. Conversation Characteristic Values that Cannot Be Set for Half-Duplex Conversations 40
5. Conversation Characteristic Values that Cannot Be Set for Full-Duplex Conversations 40
6. Conversation Queues—Associated Calls and Send-Receive Modes . . . 45
7. Calls Returning CM_OPERATION_INCOMPLETE 48
8. Incompatible conversation_security_type and required_user_name_type Values 52
9. Possible Take-Commit Notifications for Half-Duplex Conversations . . . 56
10. Possible Take-Commit Notifications for Full-Duplex Conversations 57
11. Responses to Take-Commit and Take-Backout Notifications 60
12. Responses to the CM_JOIN_TRANSACTION Indication 63
13. CPI Communications States for Protected Conversations 64
14. Languages Supported by Platform 108
15. CPI-C Calls and Product Implementation 109
16. List of CPI-C Calls and Their Descriptions 114
17. Full-Duplex and Half-Duplex Conversation Queues 342
18. Return Control Options 347
19. List of SNA Server/6000 Extension Calls for CPI Communications . . . 394
20. CICS Pseudonym Files for Supported Languages 409
21. Defaults and Allowed Values for the CICS PARTNER Resource 411
22. General Requirements for CPI Communications Calls on MVS 418
23. CPI Communications Pseudonym Files on MVS 419
24. Symptom String for Product-Specific Errors on MVS 421
25. Reason Codes for Product-Specific Errors on MVS 422
26. Location (OS/2 Subdirectory) of Pseudonym Files and Link Edit Files . 438
27. Values Returned in the REXX RC Variable for Communications Manager 442
28. An Entry of Communications Manager CPI Communications Side Information 445
29. List of Communications Manager System Management Calls 454
30. Entry Structure for the Communications Manager Extract_CPIC_Side_Information Call 458
31. Extended Entry Structure for the Communications Server Call 458
32. Entry Structure for the Communications Manager Set_CPIC_Side_Information Call 460
33. Extended Entry Structure for the Communications Server Call 461
34. Entry Structure for the Communications Server Define_TP Call 464
35. List of Communications Manager Conversation Calls 471
36. List of Communications Manager Transaction Program Control Calls . 479
37. Additional Communications Manager Characteristics Initialized following CMACCP or CMACCI 487
38. Additional Communications Manager Characteristics Initialized following CMINIT 491
39. Additional Communications Manager Characters Translated between ASCII and EBCDIC 495
40. Communications Manager Variable and Field Types and Lengths . . . 496
41. Description of OS/400 Communications Side Information Object 518

© Copyright IBM Corp. 1996, 1998

42. Summary of CPI Communications Pseudonym Files 530
43. Contents of a CMS Communications Directory File 533
44. Overview of VM/ESA Extension Routines 542
45. VM/ESA Variables/Characteristics and Their Possible Values 576
46. VM/ESA Variable Types and Lengths 577
47. VM/ESA Security Characteristics and Their Default Values 579
48. Personal Communications V4.1 for Windows 95 Publications 590
49. Client Support of CPI-C Functions 597
49. Header Files and Libraries for CPI-C 597
50. List of CPI-C Communications System Management Calls 601
51. Extended Entry Structure for the CPI-C Communications Call 605
52. Entry Structure for the CPI-C Communications Set_CPIC_Side_Information Call 607
53. Extended Entry Structure for the Communications Server Call 608
54. List of CPI-C Communications Conversation Calls 610
55. List of CPI-C Communications Transaction Program Control Calls . . . 618
56. Additional CPI-C Communications Characteristics Initialized following CMACCP or CMACCI 626
57. Additional CPI-C Communications Characteristics Initialized following CMINIT 628
58. CPI-C Communications Variable and Field Types and Lengths 632
59. Variables/Characteristics and Their Possible Values 642
60. Character Sets T61String, 01134, and 00640 647
61. Variable Types and Lengths 650
62. Fields in the Program Binding 658
63. Secondary Information Types and Associated Return Codes 679
64. Range of Condition Codes for Different Secondary Information Types . 680
65. CPI Communications-Defined Secondary Information 681
66. Examples of LU 6.2 CRM-Specific Secondary Information 692
67. Examples of OSI TP CRM-Specific Secondary Information 692
68. Examples of Implementation-Related Secondary Information 693
69. States and Transitions for CPI Communications Calls on Half-Duplex Conversations 704
70. States and Transitions for Protected Half-Duplex Conversations (CPIRR) 710
71. States and Transitions for Protected Half-Duplex Conversations (X/Open TX) 711
72. States and Transitions for CPI Communications Calls on Full-Duplex Conversations 718
73. States and Transitions for Protected Full-Duplex Conversations (CPIRR) 723
74. States and Transitions for Protected Full-Duplex Conversations (X/Open TX) 724
75. Relationship of LU 6.2 Verbs to CPI Communications Calls (Part 1) . . 729
76. Relationship of LU 6.2 Verbs to CPI Communications Calls (Part 2) . . 730
77. Relationship of LU 6.2 Verbs to CPI Communications Calls (Part 3) . . 731
78. Relationship of LU 6.2 Verbs to CPI Communications Calls (Part 4) . . 732
79. Relationship of LU 6.2 Verbs to CPI Communications Calls (Part 5) . . 733
80. Relationship of LU 6.2 Verbs to CPI Communications Calls (Part 6) . . 734
81. Parameter Type Differences 736
82. Sample Calls for CNSI 740
83. Fields in Complex Attributes 742
84. OSI TP Service Functional Units and Corresponding Conformance Classes 754
85. OSI TP Profiles and Corresponding Conformance Classes 754
86. Conformance Class Requirements—Calls 755

87. Conformance Class Requirements—Characteristics, Variables, and Values 757

Notices

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

Portions of this publication are derived from the CPI-C 2.1 Specification (SC31-6180). IBM authorizes any developer, under IBM's copyrights, to use and reproduce the materials in the CPI-C 2.1 Specification ("Materials"): to implement CPI Communications in the developer's products and to describe the interface in supporting publications for those products. The foregoing authorization does not apply to material which appears in this publication but not in the referenced CPI-C Specification. X/Open has authorized any developer to do the same with respect to any Materials in this publication in which X/Open holds the copyright.

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

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

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
208 Harbor Drive
Stamford, Connecticut
USA 06904-2501

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.

This document is not intended for production use and is furnished as is without any warranty of any kind, and all warranties are hereby disclaimed including the warranties of merchantability and fitness for a particular purpose.

© Copyright IBM Corp. 1996, 1998

Programming Interface Information

This manual is intended to help the customer program with CPI Communications. This book documents General-Use Programming Interface and Associated Guidance Information provided by CPI Communications as implemented in AIX, CICS/ESA, IMS/ESA, MVS/ESA, Networking Services for Windows, OS/2, OS/400, VM/ESA, and Windows 95.

General-Use programming interfaces allow the customer to write programs that obtain the services of CPI Communications as implemented in AIX, CICS/ESA, IMS/ESA, MVS/ESA, Networking Services for Windows, OS/2, OS/400, VM/ESA, and Windows 95.

Trademarks and Service Marks

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

AD/Cycle
Advanced Peer-To-Peer Networking
AIX
Application System/400
APPN
AS/400
BookManager
CICS/ESA
C/2
C/400
COBOL/2
COBOL/400
Common User Access
CUA
Enterprise Systems Architecture/370
Enterprise Systems Architecture/390
ESA/370
ESA/390
FORTRAN/400
IBM
ILE
IMS/ESA
MVS/ESA
Library Reader
MVS/SP
Operating System/2
Operating System/400
OS/2
OS/400
Presentation Manager
PrintManager
RISC System/6000
RPG/400
SAA
Systems Application Architecture
SystemView
System/370
System/390
Virtual Machine/Enterprise Systems Architecture
VM/ESA
VTAM

The following terms, denoted by a double asterisk (**) on the first occurrence in this publication, are trademarks of other companies:

Trademark	***Owned by***
DCE	Open Software Foundation
Windows	Microsoft Corporation
NDS	Novell Corporation
Windows 95	Microsoft Corporation
Windows NT	Microsoft Corporation
NWSAA	Novell
IWSAA	Novell

Acknowledgements

IBM wishes to express its appreciation for the teamwork of the CPI-C Implementers' Workshop (CIW) in defining CPI-C 2.0 and 2.1. Further, IBM gratefully acknowledges X/Open for its assistance, review, and refinement of CPI-C.

© Copyright IBM Corp. 1996, 1998

Part 1. CPI-C 2.0 Architecture

Chapter 1. Introduction 5
CPI-C and the Conversational Model 5
Who Should Read This Book 5
What Is New in This Book 6
Relationship to CPI-C 2.1 Specification 6
Relationship to Products 7
Additional Information Sources 8
Previous Editions of this Reference 8
Related APPC/LU 6.2 Publications 8
CD-ROM 9
Solution Developers Organization 9
Online Information—CPI-C, APPC, and APPN 9
CompuServe 9
OS/2 BBS 9
Internet 10
Functional Levels of CPI Communications 10
CPI-C 1.0 10
CPI-C 1.1 10
X/Open CPI-C 10
CPI-C 1.2 10
CPI-C 2.0 11
X/Open CPI-C 2.0 11
CPI-C 2.1 11
Call Table for Functional Levels of CPI-C 12
Naming Conventions—Calls, Characteristics, Variables, and Values 13

Chapter 2. CPI Communications Terms and Concepts 17
Communication across a Network 18
Conversation Types 19
Send-Receive Modes 19
Program Partners 20
Identifying the Partner Program 20
Operating Environment 21
Node Services 22
Side Information 23
Distributed Directory 25
CPI Communications Directory Object 25
Using the Distributed Directory 26
Interaction with Side Information and Set Calls 27
Distributed Security 28
Operating System 28
Program Calls 29
Starter Set Calls 29
Establishing a Conversation 29
Multiple Conversations 30
Partner Program Names 30
Multiple Outbound Conversations 30
Multiple Inbound Conversations 31
Contexts and Context Management 32
Relationship between Contexts and Conversations 32

© Copyright IBM Corp. 1996, 1998

Relationship between Contexts and Security Parameters 32
Inbound and Outbound Conversations 32
Conversation Characteristics 33
Modifying and Viewing Characteristics 34
Characteristic Values and CRMs 39
Characteristic Values and Send-Receive Modes 40
Automatic Conversion of Characteristics 41
Automatic Data Conversion 42
Data Conversion 43
Data Buffering and Transmission 44
Concurrent Operations 44
Using Multiple Program Threads 45
Non-Blocking Operations 47
Conversation-Level Non-Blocking 48
Queue-Level Non-Blocking 49
Working with Wait Facility 49
Wait Facility Scenario 49
Using Callback Function 50
Canceling Outstanding Operations 50
Non-Blocking Calls and Context Management 50
Conversation Security 51
Program Flow—States and Transitions 52
Support for Resource Recovery Interfaces 54
Coordination with Resource Recovery Interfaces 55
Take-Commit and Take-Backout Notifications 55
The Backout-Required Condition 58
Responses to Take-Commit and Take-Backout Notifications 59
Chained and Unchained Transactions 61
Joining a Transaction 61
Superior and Subordinate Programs 63
Additional CPI Communications States 64
Valid States for Resource Recovery Calls 65
TX Extensions for CPI Communications 66

Chapter 3. Program-to-Program Communication Example Flows 67
Interpreting the Flow Diagrams 67
Starter-Set Flows 68
Example 1: Data Flow in One Direction 69
Example 2: Data Flow in Both Directions 72
Controlling Data Flow Direction 74
Example 3: The Sending Program Changes the Data Flow Direction 74
Example 4: The Receiving Program Changes the Data Flow Direction . . . 75
Verifying Receipt of Data 78
Example 5: Validation of Data Receipt 78
Reporting Errors to Partner 80
Example 6: Reporting Errors 80
Example 7: Error Direction and Send-Pending State 82
Using Full-Duplex Conversations 84
Example 8: Establishing a Full-Duplex Conversation 84
Example 9: Using a Full-Duplex Conversation 86
Example 10: Terminating a Full-Duplex Conversation 88
Using Queue-Level Non-Blocking 90
Example 11: Queue-Level Non-Blocking 90
Accepting Multiple Conversations 92

Example 12: Accepting Multiple Conversations Using Blocking Calls 92
Example 13: Accepting Multiple Conversations Using Conversation-Level Non-Blocking Calls 94
Using the Distributed Directory 96
Example 14: Using the Distributed Directory to Find the Partner Program . 96
Resource Recovery Flows 98
Example 15: Sending Program Issues a Commit 98
Example 16: Successful Commit with Conversation State Change 100
Example 17: Conversation Deallocation before the Commit Call 102

Chapter 1. Introduction

This introductory chapter:

- Explains CPI Communications and the conversational model
- Identifies the purpose and audience of this book
- Lists what is new in this book
- Explains the relationship to CPI-C 2.1 specification
- Explains the relationship to products
- Details the availability of additional information sources
- Lists the functional levels of CPI Communications
- Provides a call table with the functional levels of CPI-C
- Explains the naming conventions followed

CPI-C and the Conversational Model

CPI Communications (CPI-C) provides a cross-system-consistent and easy-to-use programming interface for applications that require program-to-program communication. The conversational model of program-to-program communication is commonly used in the industry today, and a wide variety of applications are based on this model. The model is described in terms of two applications—***speaking*** and ***listening***—hence, the term ***conversation***. A conversation is simply a logical connection between two programs that allows the programs to communicate with each other. From an application's perspective, CPI-C provides the function necessary to enable this communication.

The conversational model is implemented in two major communications protocols, Advanced Program-to-Program Communication (APPC) and Open Systems Interconnection Distributed Transaction Processing (OSI TP).[1] The APPC protocol is also referred to as logical unit type 6.2 (LU 6.2). CPI-C provides access to both APPC and OSI-TP.

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

Who Should Read This Book

This book defines CPI Communications. It is intended for programmers who want to write applications that use communications products supporting CPI Communications.

Although this book is designed as an API reference, Chapter 3, "Program-to-Program Communication Example Flows" provides examples for designing application programs using CPI Communications concepts and calls.

[1] International Standardization Organization (ISO) and International Electrotechnical Commission (IEC) standard 10026 1 – 3: *Information Technology – Open Systems Interconnection — Distributed Transaction Processing — Parts 1 – 3*.

© Copyright IBM Corp. 1996, 1998

What Is New in This Book

This book includes changes to CPI-C 2.1 architecture and changes to product implementation information.

CPI-C 2.1 Enhancements Providing Support For:

- Enhanced Cancel_Conversation
- Additional return control options
- Accept specific
- Automatic Data Conversion
- Service Transaction Programs

Product implementations updated:

- AIX to AIX SNA Server/6000 Version 3 Release 1 and Version 2, Release 1.2
- OS/2 to Communications Server for OS/2 WARP Version 4.0 and Personal Communications AS/400 and 3270 for OS/2 Version 4.1.
- IMS to IMS/ESA Version 5.

Product implementations added:

- Windows 95 environment. Support is provided for IBM eNetwork Personal Communications AS/400 and 3270 for Windows 95, and IBM eNetwork Personal Communications AS/400 for Windows 95.
- IBM eNetwork Communication Server for Windows NT 5.0, 5.01, and above
- Win95 API Client for Communication Server 5.0, 5.01, and above
- WinNT API Client for Communication Server 5.0, 5.01, and above
- OS/2 API Client for Communication Server 5.0, 5.01, and above
- Win95 API Client for Netware for SAA 2.2
- WinNT API Client for Netware for SAA 2.2
- OS/2 API Client for Netware for SAA 2.2
- Win95 API Client for IntraNetware for SAA 2.3, 3.0 and above
- WinNT API Client for IntraNetware for SAA 2.3, 3.0 and above
- OS/2 API Client for IntraNetware for SAA 2.3, 3.0 and above
- IBM eNetwork Personal Communications 4.1 for WinNT and above
- IBM eNetwork Personal Communications 4.2 for Win95 and above

Relationship to CPI-C 2.1 Specification

This book contains all the information in the *CPI-C 2.1 Specification* with these exceptions:

- Most notes to implementers are removed, since they provide no value to interface users.
- Some OSI-TP specific information is not included.
- Pseudonym files are provided on diskette.

Relationship to Products

The CPI Communications interface defines elements that are consistent across the operating environments. Preparing and running programs require the use of a communications product that implements the CPI-C interface for the desired environment.

The following environments are supported by the listed IBM products, which are described in this manual:

Environment	**Implementing Product**
AIX	SNA Server/6000 SNA Desktop/6000
CICS/ESA	CICS/ESA
IMS/ESA	IMS/ESA Transaction Manager
MVS/ESA	MVS/ESA System Product
OS/2	Communications Server for OS/2 WARP Personal Communications for OS/2 API Client for Communication Server 5.0, 5.01, and above API Client for Netware for SAA 2.2 API Client for IntraNetware for SAA 2.2, 3.0, and above
OS/400	Operating System/400
VM/ESA	VM/ESA
Windows**	IBM APPC Networking Services for Windows
Windows 95**	IBM eNetwork Personal Communications for Windows 95 Win95 API Client for Communication Server 5.0, 5.01, and above Win95 API Client for Netware for SAA 2.2 Win95 API Client for IntraNetware for SAA 2.3, 3.0, and above IBM eNetwork Personal Communications v4.2 for Windows95 and WindowNT and above
Windows NT**	IBM eNetwork Communication Server for Windows NT 5.0, 5.01, and above WinNT API Client for Communication Server 5.0, 5.01, and above WinNT API Client for Netware for SAA 2.2 WinNT API Client for IntraNetware for SAA 2.3, 3.0, and above IBM eNetwork Personal Communications v4.1 for WinNT IBM eNetwork Personal Communications v4.2 for Windows95 and WindowNT and above

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

See the appropriate product chapters in “Part 3. CPI-C 2.1 Implementation Specifics” for a list of product documentation currently available.

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

CPI-C definition in its syntax or semantics, the text states that fact and is printed in green as is this sentence. In addition, the sentence has a *g* in the margin.

Additional Information Sources

This section lists additional resources for CPI Communications information. If you need information about using CPI Communications in a specific operating environment, please see the appropriate product chapter in "Part 3. CPI-C 2.1 Implementation Specifics."

Previous Editions of this Reference

Table 1 lists the previous editions of this book still available, the order numbers, and the versions of the implementing product described.

Table 1. Previous Editions of the CPI-C Reference

Edition	Number	Products
8th Edition SC26-4399-07	ST01-0223	AIX SNA Server/6000, Version 2 Release 1.1 IMS/ESA, Version 4
7th Edition SC26-4399-06	ST00-8760	AIX SNA Services/6000, Release 1.2.1 VM/ESA, Release 2 MVS/ESA, Release 4.3
6th Edition SC26-4399-05	ST00-6053	MVS/ESA, Release 4.2.2 VM/ESA, Release 1.1 AIX SNA Services/6000, Release 1.2.1 IBM Extended Services for OS/2, Version 1, R0 MVS/ESA, Version 4, Release 3 IBM SAA Networking Services/2, Version 1 Networking Services DOS, Version 1 Operating System/400, Version 2, Release 2
5th Edition SC26-4399-04	ST00-5632	CICS/ESA, Version 3, Release 2 IMS/ESA, Version 3, Release 2 MVS/ESA, Version 4, Release 2
4th Edition SC26-4399-03	ST00-5002	Operating System/400, Version 2, Release 1

Related APPC/LU 6.2 Publications

The following publications contain introductory level information on CPI-C or APPC:

- *Systems Network Architecture Technical Overview*, GC30-3073
- *CPI-C Programming in C: An Application Developer's Guide to APPC* (John Q. Walker II and Peter J. Schwaller. McGraw-Hill, Inc., 1994, ISBN 0-07-911733-3. Paperback, with diskette), SR28-5597

The following publications contain detailed technical information on CPI-C or APPC:

- *CPI Communications: CPI-C 2.1 Specification*, SC31-6180
- *SNA Formats*, GA27-3136
- *SNA LU 6.2 Reference: Peer Protocols*, SC31-6808
- *Systems Network Architecture: Sync Point Services Architecture Reference*, SC31-8134

- *SNA Transaction Programmer's Reference Manual for LU Type 6.2*, GC30-3084
- *Multiplatform APPC Configuration Guide*, GG24-4485

Note: Product-specific information about using CPI Communications in each operating environment can be found in the appropriate chapters in "Part 3. CPI-C 2.1 Implementation Specifics."

CD-ROM

This publication is also available as a softcopy book. The softcopy book is on an electronic bookshelf and is part of the *IBM Networking Systems Softcopy Collection Kit* (SK2T-6012) on compact disk read-only memory (CD-ROM).

You can view and search softcopy books by using BookManager* READ products or by using the IBM Library Reader* product included on each CD-ROM. For more information on CD-ROMs and softcopy books, see *IBM Online Libraries: Softcopy Collection Kit User's Guide* (GC28-1700) and BookManager READ documentation.

Solution Developers Organization

The Solution Developers Organization (SDO) is open to independent software vendors (ISVs) who are developing, or planning to develop, APPC and/or CPI-C support in their products. The SDO provides you with technical, business, and marketing services related to the development and promotion of APPC support in products. For full details, please see Appendix H, "Solution Developers Program - Enterprise Communications Partners in Development" on page 763.

Online Information—CPI-C, APPC, and APPN

You can access helpful online information on CPI-C, APPC, and APPN and utilities through CompuServe, the OS/2 BBS, and the Internet. You can get up-to-date information on CPI-C, access sample programs and tools, ask questions, and provide feedback on CPI-C and related IBM products. You can post questions about CPI-C programming and architecture, configuration, and about using APPC with any of IBM's products like CICS, Communications Manager/2, APPC Networking Services for Windows, AIX SNA Server/6000, AS/400, VTAM/NCP, MVS, and VM.

CompuServe

IBM's APPC Market Enablement team maintains the APPC/APPN Forum (GO APPC) on CompuServe. More than a dozen question-and-answer sections exist as well as hundreds of sample programs, utilities, and technical papers.

To get a free introductory membership, call CompuServe. In the US and Canada, the phone number is 800-848-8199; in the UK, 0800-289-378; in Germany, 0130-37-32. Elsewhere the phone number is 1-614-457-0802. Be sure to ask for representative 337.

OS/2 BBS

The OS/2 BBS APPC FORUM is available on the IBM Information Network (IIN) and the IBMLink TalkLink facility. In the US, the phone number is 1-800-547-1283; in Canada, 1-800-465-7999 (ext 228). Elsewhere, contact your local IBM representative.

Internet

A wide variety of APPC announcements, utilities, and CPI-C sample programs are available from the Internet via the URL:

`http://www.raleigh.ibm.com/aac/aachome.htm`

Functional Levels of CPI Communications

CPI Communications is an evolving interface, embracing functions to meet the growing demands from different application environments and to achieve openness as an industry standard for communications programming. This section contains a brief history of each functional level of CPI Communications and the additional function provided.

CPI-C 1.0

CPI Communications, introduced in 1987, provided the standard base for conversational communications:

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

CPI-C 1.1

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

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

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

X/Open CPI-C

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

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

CPI-C 1.2

CPI-C 1.0 was designed to provide a consistent programming interface for communications programming. However, each of its derivatives, namely CPI-C 1.1 and X/Open CPI-C, provided different levels of function. CPI-C 1.2, documented in the first edition of *CPI Communications*, SC31-6180-00, consolidated CPI-C 1.1 and the X/Open extensions, providing function in four areas:

- Support for non-blocking calls—incorporation of X/Open calls

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

CPI-C 2.0

CPI-C 2.0, completed by the CPI-C Implementers' Workshop (CIW) in 1994, provides enhancements to some CPI-C 1.2 functions, as well as offering several new functions:

- Support for full-duplex conversations and expedited data
- Enhanced support for non-blocking processing with the addition of queue-level processing and a callback function
- Support for OSI TP applications
- Support for use of a distributed directory
- Support for use of a distributed security service
- Support for secondary information to determine the cause of a return code
- Definition of conformance classes

X/Open CPI-C 2.0

X/Open CPI-C 2.0 enhances and updates X/Open CPI-C to the CPI-C 2.0 level with the following exceptions:

- Supports only C and COBOL programming languages.
- Does not include distributed directory support. Specifically, the Set_Partner_ID and Extract_Partner_ID calls are not included.
- In the COBOL pseudonym file, X/Open uses COMP-5 for integers—whereas, CIW CPI-C 2.0 uses COMP-4 for integers.

CPI-C 2.1

CPI-C 2.1, completed by the CPI-C Implementors' Workshop (CIW) in 1995, provides the following functions:

- Support for automatic data conversation
- Support for the ability to accept conversations with a specific TP name
- Support for the specification of service TP names
- Support for additional return control options
- Enhancements to the Cancel_Conversations call to allow determination of cancelled operations.

Call Table for Functional Levels of CPI-C

Table 2 lists the calls defined for the different versions of CPI Communications. An X is used to indicate that the call was part of a specific version.

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

Call Name	CPI-C 1.0	CPI-C 1.1	X/Open CPI-C	CPI-C 1.2	CPI-C 2.0	X/Open CPI-C 2.0	CPI-C 2.1
Starter Set							
Accept_Conversation	X	X	X	X	X	X	X
Allocate	X	X	X	X	X	X	X
Initialize_Conversation	X	X	X	X	X	X	X
Receive	X	X	X	X	X	X	X
Send_Data	X	X	X	X	X	X	X
Advanced Function for synchronization and control							
Confirm	X	X	X	X	X	X	X
Confirmed	X	X	X	X	X	X	X
Deferred_Deallocate					X	X	X
Flush	X	X	X	X	X	X	X
Include_Partner_In_Transaction					X	X	X
Prepare					X	X	X
Prepare_To_Receive	X	X	X	X	X	X	X
Receive_Expedited_Data					X	X	X
Request_To_Send	X	X	X	X	X	X	X
Send_Error	X	X	X	X	X	X	X
Send_Expedited_Data					X	X	X
Test_Request_To_Send_Received	X	X	X	X	X	X	X
Advanced Function for modifying conversation characteristics:							
Set_AE_Qualifier					X	X	X
Set_Allocate_Confirm					X	X	X
Set_AP_Title					X	X	X
Set_Application_Context_Name					X	X	X
Set_Begin_Transaction					X	X	X
Set_Confirmation_Urgency					X	X	X
Set_Conversation_Security_Password			X	X	X	X	X
Set_Conversation_Security_Type			X	X	X	X	X
Set_Conversation_Security_User_ID			X	X	X	X	X
Set_Conversation_Type	X	X	X	X	X	X	X
Set_Deallocate_Type	X	X	X	X	X	X	X
Set_Fill	X	X	X	X	X	X	X
Set_Initialization_Data					X	X	X
Set_Join_Transaction					X	X	X
Set_Log_Data	X	X	X	X	X	X	X
Set_Mode_Name	X	X	X	X	X	X	X
Set_Partner_ID					X		X
Set_Partner_LU_Name	X	X	X	X	X	X	X
Set_Prepare_Data_Permitted					X	X	X
Set_Prepare_To_Receive_Type	X	X	X	X	X	X	X
Set_Receive_Type	X	X	X	X	X	X	X
Set_Return_Control	X	X	X	X	X	X	X
Set_Send_Receive_Mode					X	X	X
Set_Send_Type	X	X	X	X	X	X	X
Set_Sync_Level							
CM_NONE	X	X	X	X	X	X	X
CM_CONFIRM	X	X	X	X	X	X	X
CM_SYNC_POINT		X		X	X	X	X
CM_SYNC_POINT_NO_CONFIRM					X	X	X
Set_TP_Name	X	X	X	X	X	X	X
Set_Transaction_Control					X	X	X

Table 2 (Page 2 of 2). Versions of CPI Communications							
Call Name	**CPI-C 1.0**	**CPI-C 1.1**	**X/Open CPI-C**	**CPI-C 1.2**	**CPI-C 2.0**	**X/Open CPI-C 2.0**	**CPI-C 2.1**
Advanced Function for examining information about the conversation and CRM:							
Extract_AE_Qualifier					X	X	X
Extract_AP_Title					X	X	X
Extract_Application_Context_Name					X	X	X
Extract_Conversation_Context				X	X		X
Extract_Conversation_State		X		X	X	X	X
Extract_Conversation_Type	X	X	X	X	X	X	X
Extract_Initialization_Data					X	X	X
Extract_Maximum_Buffer_Size				X	X	X	X
Extract_Mode_Name	X	X	X	X	X	X	X
Extract_Partner_ID					X		X
Extract_Partner_LU_Name	X	X	X	X	X	X	X
Extract_Secondary_Information					X	X	X
Extract_Security_User_ID			–[2]	X	X	X	X
Extract_Send_Receive_Mode					X	X	X
Extract_Sync_Level	X	X	X	X	X	X	X
Extract_TP_Name			X	X	X	X	X
Extract_Transaction_Control					X	X	X
Advanced Function for non-blocking operations:							
Cancel_Conversation			X	X	X	X	X
Set_Processing_Mode			X	X	X	X	X
Set_Queue_Callback_Function					X	X	X
Set_Queue_Processing_Mode					X	X	X
Wait_For_Completion					X	X	X
Wait_For_Conversation			X	X	X	X	X
Advanced Function for accepting multiple conversations:							
Accept_Incoming				X	X	X	X
Initialize_For_Incoming				X	X	X	X
Release_Local_TP_Name				X	X	X	X
Specify_Local_TP_Name			X	X	X	X	X
Advanced Function for data conversion:							
Convert_Incoming			X	X	X	X	X
Convert_Outgoing			X	X	X	X	X
Extract_Mapped_Initialization_Data							X
Receive_Mapped_Data							X
Send_Mapped_Data							X
Set_Mapped_Initialization_Data							X

Naming Conventions—Calls, Characteristics, Variables, and Values

Pseudonyms for the actual calls, characteristics, variables, states, and characteristic values comprising CPI Communications are used throughout this book to enhance understanding and readability.

Where possible, underscores (_) and complete names are used in the pseudonyms. Any phrase in the book that contains an underscore is a pseudonym.

[2] X/Open CPI-C provides an Extract_Conversation_Security_User_ID call that provides similar function.

For example, Send_Data is the pseudonym for the program call CMSEND, which is used by a program to send information to its conversation partner.

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

- **Calls** are shown in all capital letters. Each underscore-separated portion of a call's pseudonym begins with a capital letter. For example, Accept_Conversation is the pseudonym for the actual call name CMACCP.
- **Characteristics** and **variables** used to hold the values of characteristics are in italics (for example, *conversation_type*) and contain no capital letters except those used for abbreviations (for example, *TP_name*).

  In most cases, the parameter used on a call, which corresponds to a program variable, has the same name as the conversation characteristic. Whether a name refers to a parameter, a program variable, or a characteristic is determined by context. In all cases, the value used for the three remains the same.
- **Values** used for characteristics and variables appear in all uppercase letters (such as CM_OK) and represent actual integer values that will be placed into the variable. For a list of the integer values that are placed in the variables, see Table 59 on page 642 in Appendix A, "Variables and Characteristics."
- **States** are used to determine the next set of actions that can be taken in a conversation. States begin with capital letters and appear in bold type, such as **Reset** state. Bold is also used to denote the **Backout-Required** condition.
- **Queues** are used to group related CPI Communications calls. Queue names begin with capital letters. The parts of a queue name are connected with a hyphen.

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

- "Set_Return_Control (CMSRC)" contains the syntax and semantics of the variables used for the call. It explains that the real name of the program call for Set_Return_Control is CMSRC and that CMSRC has a parameter list of *conversation_ID*, *return_control*, and *return_code*.
- Appendix A, "Variables and Characteristics" provides a complete description of all variables used in the book and shows that the *return_control* variable, which goes into the Set_Return_Control call as a parameter, is a 32-bit integer. This information is provided in Table 61 on page 650.
- Table 59 on page 642 in Appendix A, "Variables and Characteristics" shows that CM_IMMEDIATE is defined as having an integer value of 1. CM_IMMEDIATE is placed into the *return_control* parameter on the call to CMSRC.
- Finally, the *return_code* value CM_OK, which is returned to the program on the CMSRC call, is defined in Appendix B, "Return Codes and Secondary Information." CM_OK means that the call completed successfully.

Notes:

1. Pseudonym value names are not actually passed to CPI Communications as a string of characters. Instead, the pseudonyms represent integer values that are passed on the program calls. The pseudonym value names are used to aid readability of the text. Similarly, programs should use translates and equates (depending on the language) to aid the readability of the code. In the above example, for instance, a program equate could be used to define CM_IMMEDIATE as meaning an integer value of 1. The actual program code would then read as described above—namely, that *return_control* is replaced with CM_IMMEDIATE. The end result, however, is that an integer value of 1 is placed into the variable.
2. Section "Programming Language Considerations" on page 111 in "Set_Return_Control (CMSRC)" provides information on system files that can be used to establish pseudonyms for a program.

Chapter 2. CPI Communications Terms and Concepts

CPI Communications provides a consistent application programming interface for applications that require program-to-program communication. The interface provides access to a rich set of interprogram services, including:

- Sending and receiving data
- Synchronizing processing between programs
- Notifying a partner of errors in the communication

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

© Copyright IBM Corp. 1996, 1998

Communication across a Network

Figure 1 illustrates the logical view of a sample network. It consists of three **communication resource managers**[3] (CRMs): CRM X, CRM Y, and CRM Z. Each CRM has two **logical connections** with two other CRMs; the logical connections are shown as the gray portions of Figure 1 and enable communication between the CRMs. The network shown in Figure 1 is a simple one. In a real network, the number of CRMs and logical connections between the CRMs can be in the tens of thousands or higher.

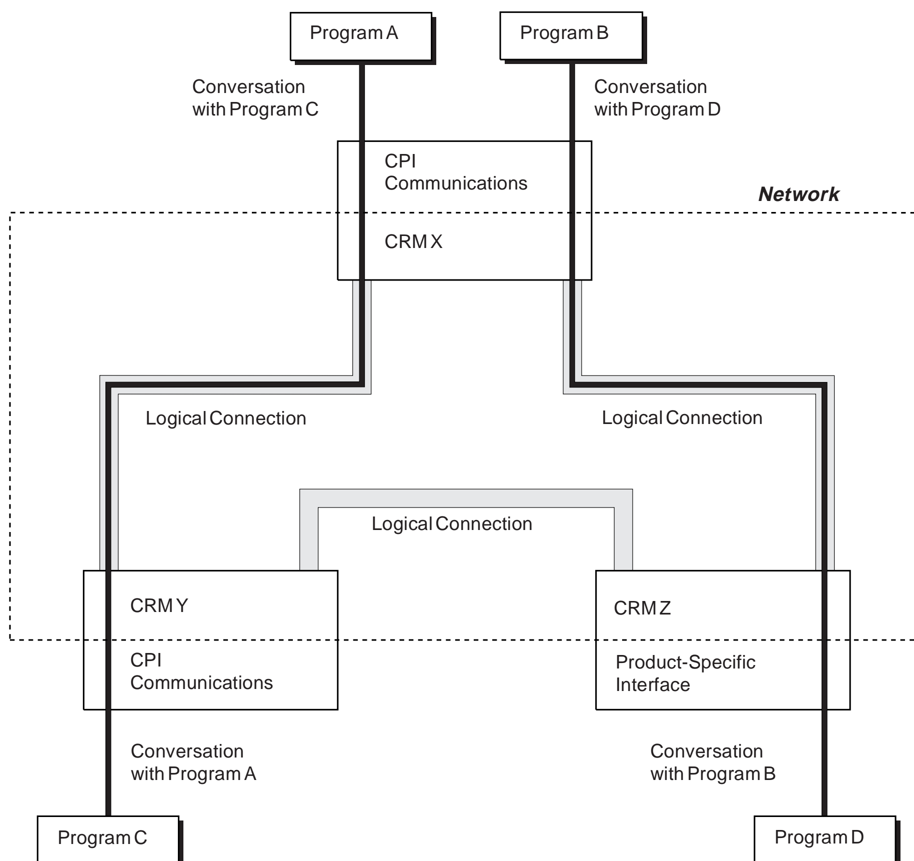


Figure 1. Programs Using CPI Communications to Converse through a Network

The CRMs and the logical connections shown in Figure 1 are generic representations of real networks. If this were an SNA network, the CRMs would be referred to as **logical units** of type 6.2 and the logical connections would be **sessions**. In an OSI network, CRMs are called **application-entities** and the logical connections are **associations**. The physical network, which consists of nodes (processors) and data links between nodes, is not shown in Figure 1 because a program using CPI Communications does not “see” these resources. A program uses the logical network of CRMs, which in turn communicates with and uses the physical network. This manual discusses CRMs of type LU 6.2 and type OSI TP.

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

Conversation Types

Just as two CRMs communicate using a logical connection, two programs exchange data using a **conversation**. For example, the conversation between Program A and Program C is shown in Figure 1 as a single bold line between the two programs. The line indicating the conversation is shown on top of the logical connection because a conversation allows programs to communicate "over" the logical connection between the CRMs.

CPI Communications supports two types of conversations:

- **Mapped** conversations allow programs to exchange arbitrary **data records** in data formats agreed upon by the application programmers. CPI Communications supports both the traditional pass through method, where the application program encodes and decodes the data, and external routines that encode and decode user data.
- **Basic** conversations allow programs to exchange data in a standardized format. This format is a stream of data containing 2-byte logical length fields (referred to as LLs) that specify the amount of data to follow before the next length field. The typical data pattern is "LL, data, LL, data." Each grouping of "LL, data" is referred to as a **logical record**.

  Notes:

  1. Because of the detailed manipulation of data and resulting complexity of error conditions, the use of basic conversations is intended for programs with specialized requirements. A more complete discussion of basic and mapped conversations is provided in the "Usage Notes" section of "Send_Data (CMSEND)" on page 249.
  2. Because OSI TP CRMs do not exchange the conversation characteristic that determines whether a conversation will be mapped or basic, the remote application must also issue a Set_Conversation_Type call when basic conversations are being used, to override the default value of CM_MAPPED_CONVERSATION for the *conversation_type* conversation characteristic.

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

Send-Receive Modes

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

- **Half-duplex**—Only one of the programs has **send control,** the right to send data, at any time. Send control must be transferred to the other program before that program can send data.
- **Full-duplex**—Both programs can send and receive data at the same time. Thus, both programs have send control.

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

For further information, see “Characteristic Values and Send-Receive Modes” on page 40.

Program Partners

Two programs involved in a conversation are called **partners** in the conversation. If a CRM-CRM logical connection exists, or can be created, between the nodes containing the partner programs, two programs can communicate through the network with a conversation.

The terms local and remote are used to differentiate between different ends of a conversation. If a program is being discussed as **local**, its partner program is said to be the **remote** program for that conversation. For example, if Program A is being discussed, Program A is the local program and Program C is the remote program. Similarly, if Program C is being discussed as the local program, Program A is the remote program. Thus, a program can be both local and remote for a given conversation, depending on the circumstances.

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

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

Note: A CPI Communications program may establish a conversation with a program that is using a product-specific programming interface for a particular environment and not CPI Communications. The conversation between Program B and Program D in Figure 1 is an example of such a situation. Some restrictions may apply in this situation, since CPI Communications does not support all available network functions.

Identifying the Partner Program

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

- **Program-supplied**
  The program can supply the destination information directly.
- **Side information**
  The program can use data contained in local **side information**. The side information is accessed using an 8-byte **symbolic destination name** or *sym_dest_name*, which identifies the partner program.
- **Distributed directory**
  The program can use a **directory object** contained in a **distributed directory**. The directory object represents a particular installation of a program and is

identified by a variable-length **distinguished name** (DN). Programs not having a DN for their partner program can search the directory for objects containing a **program function identifier** (PFID). The PFID is a globally-unique identifier for the function provided by the program.

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

- Program-supplied information may require recompilation if the address of the partner program changes.
- Use of *sym_dest_name* allows only a small name space of locally-defined names.
- Side information requires local administration on each system.
- Movement of a program may result in update of side information on multiple systems.
- Use of a distributed directory enables destination information to be stored in a single location and accessed by multiple programs.
- As with the *sym_dest_name*, a DN identifies a particular installation of a program and destination information may not be known at program development time. For example, shrink-wrapped programs will not have access to the DN when distributed. Use of the PFID solves this problem by allowing a run-time search of the directory for all program installation objects with a particular functionality.
- "Side Information" on page 23 and "Distributed Directory" on page 25 discuss how existing CPI Communications programs using side information can be migrated to use the distributed directory by using the *sym_dest_name* to identify a side-information entry containing a DN.

Operating Environment

Figure 2 on page 22 gives a more detailed view of Program A's operating environment. As in Figure 1, the bold black line shows the conversation Program A has established with its partner program. The new line between the program and CPI Communications represents Program A's use of program calls to communicate with CPI Communications. The different types of CPI Communications calls are discussed in "Program Calls" on page 29.

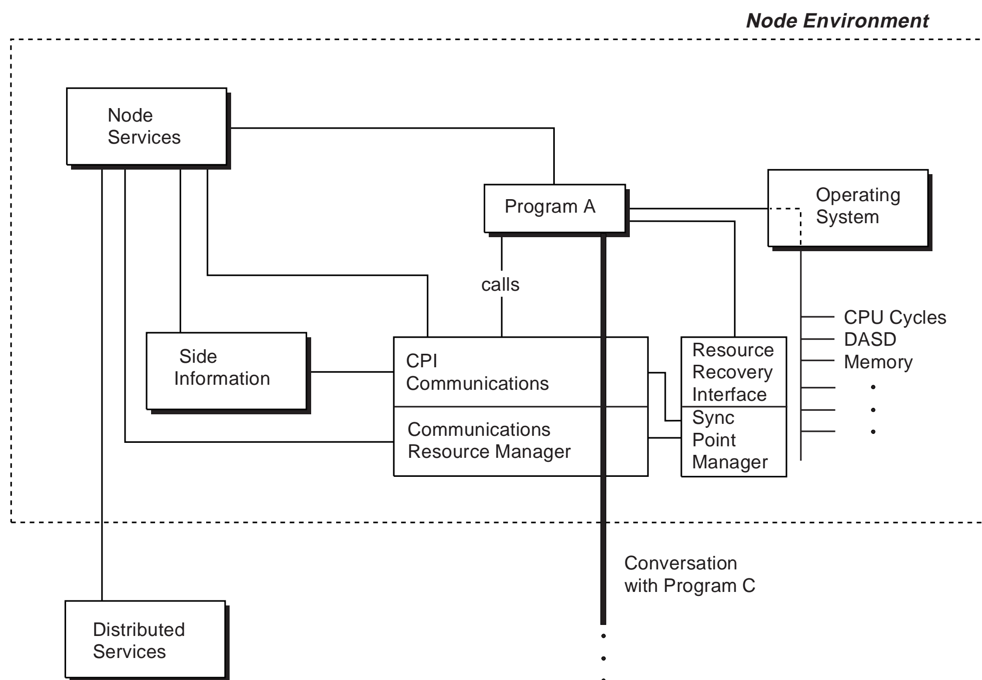


Figure 2. Operating Environment of CPI Communications Program

In addition to the new line with CPI Communications, Figure 2 also shows Program A using several other generic elements:

- Node services
- Side information
- Distributed services
  - Distributed directory
  - Distributed security
- Resource recovery interface
- Operating system

These elements are further discussed within this chapter.

Node Services

Node services represents a number of "utility" functions within the local system environment that are available for CPI Communications and other programming interfaces. These functions are not related to the actual sending and receiving of CPI Communications data, and specific implementations differ from product to product. Node services includes the following general functions:

- Setting and accessing of side information

  This function is required to set up the initial values of the side information and allow subsequent modification. It does not refer to individual program modification of the program's copy of the side information using Set calls, as described in "Conversation Characteristics" on page 33. (Refer to specific product information for details.)

- Setting and accessing of distributed directory information

  Node services is used by programs requiring direct interaction with a distributed directory. For example, a program might need to perform a search of the directory in order to determine the correct destination information. Once the

correct object has been determined by making calls to node services, the program can pass the program destination information to CPI Communications. Alternatively, the program may pass a DN or PFID to CPI Communications and CPI Communications will access the directory (using node services) to retrieve the destination information for the program. For more information, see "Distributed Directory" on page 25.

- Program-startup processing

  A program is started either by receipt of notification that the remote program has issued an Allocate call for the conversation (discussed in greater detail in "Starter-Set Flows" on page 68) or by local (operator) action. In either case, node services sets up the data paths and operating environment required by the program, validates and establishes security parameters under which the program will execute, and then allows the program to begin execution. In the former case, node services receives the notification, retrieves the name of the program to be started and any access security information included in the conversation startup request, and then proceeds as if starting a program by local action.

- Program-termination processing (both normal and abnormal)

  The program should terminate all conversations before the end of the program. However, if the program does not terminate all conversations, node services will abnormally deallocate any dangling conversations.

  Please see the appropriate product chapter in "Part 3. CPI-C 2.1 Implementation Specifics" on page 373 for details on how each operating environment detects and deallocates dangling conversations.

- Providing support for programs with multiple partners

  As is discussed in greater detail in "Multiple Conversations" on page 30, some programs do work on behalf of multiple partners. Each partner is represented by a **context**, or collection of logical attributes. Node services provides the necessary function and support (through system interfaces) to allow the program to manage different partner contexts. See "Contexts and Context Management" on page 32 for more information.

- Acquiring and validating access security information

  Node services provides interfaces for CRMs both to acquire and to validate access security information on behalf of a user. In a distributed system, node services may use a distributed security service that provides authentication services to create or validate the access security information. See "Conversation Security" on page 51 for more information.

Side Information

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

The possible information specific to a LU 6.2 CRM is:

- ***partner_LU_name***

  Indicates the name of the LU where the partner program is located. This LU name is any name for the remote LU recognized by the local LU for the purpose of allocating a conversation.

The possible information specific to an OSI-TP CRM is:

- ***AP_title***

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

- ***AE_qualifier***

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

- ***application_context_name***

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

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

- ***TP_name***

  Specifies the name of the remote program. *TP_name* stands for "transaction program name." In this manual, *transaction program*, *application program*, and *program* are synonymous, all denoting a program using CPI Communications.

- ***mode_name***

  Used to designate the properties of the logical connection that will be established for the conversation. The properties include, for example, the class of service to be used on the conversation. The system administrator defines a set of mode names used by the local CRM to establish logical connections with its partners. An LU 6.2 CRM always has the following modes defined for application usage: #BATCH, #BATCHSC, #INTER, and #INTERSC.

- ***distinguished_name***

  Indicates a DN that can be used to access destination information in a distributed directory. The information retrieved from the distributed directory will be used to establish the conversation. Associated attributes of *directory_syntax* and *directory_encoding* may be present. If a *distinguished_name* is present, only the security information from the entry (*conversation_security_type*, *security_user_ID*, and *security_password*) will be used. All other destination information from this side information entry will be ignored.

- ***conversation_security_type***

  Specifies the type of access security information to be included in the conversation startup request. See "Conversation Security" on page 51 for more information.

- ***security_user_ID***

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

- ***security_password***

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

Programs not wanting to use side information can specify a *sym_dest_name* of blanks on the Initialize_Conversation call. For more information, see “Initialize_Conversation (CMINIT)” on page 200.

On VM, if a corresponding entry is not found in the side information table, the name provided in *sym_dest_name* will be used as the partner *TP_name*.

Distributed Directory

A distributed directory is a service that enables information to be stored in a single location. It is referred to as “distributed” because the information can be accessed from multiple locations in a network using local directory interfaces. The local directory interface (part of node services) handles the communications and information flows required to retrieve the requested information from the directory. Information is stored in the directory by placing it in a directory object. Directory objects may contain many different pieces of information.

Figure 3 shows Program A interacting directly with a local directory interface. When Program A provides a name to node services (1), node services accesses the distributed directory (2 and 3). The retrieved object is then returned to the program (4).

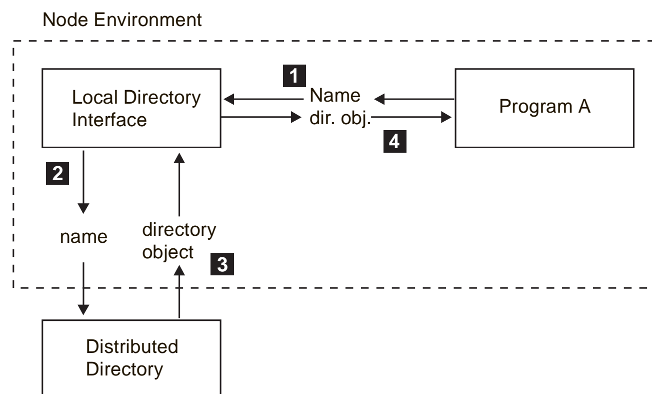


Figure 3. Generic Program Interaction with a Distributed Directory

CPI Communications Directory Object

A directory object that contains the destination information for a single installation of a partner program is referred to as a **program installation object**. A program installation object includes the following information:

- **Program Function Identifier (PFID)**
  A PFID uniquely identifies the function provided by a program. This allows programs installed on multiple systems, with different DNs, to be recognized as providing the same function. For example, multiple installations of a distributed mail application might all have the same PFID. See “Program Function Identifier (PFID)” on page 656 for a more detailed discussion of the PFID.

Note: On distributed directories supporting attribute types, the PFID attribute type is uniquely identified with a registered ISO object identifier of 1.3.18.0.2.4.13.

- **Program Binding**
  The program binding contains information required by a partner program to establish a conversation with the program. Because multiple CRMs can be used to reach the same program installation, there may be more than one program binding in a single directory object. Each program binding contains the following information:
  - local address (TP_name)
  - mode (mode_name)
  - partner_principal_name—identifies the principal name used by the remote CRM for authentication of conversation startup requests
  - required_user_name_type—identifies the type of user name required for access to the partner program
  - CRM-specific information

    For LU 6.2:

    - partner_LU_name

    For OSI TP:

    - AE_qualifier
    - AP_title
    - application_context_name

Notes:

1. On distributed directories supporting attribute types, the CPI Communications program binding attribute is uniquely identified with a registered ISO object identifier of 1.3.18.0.2.4.14.

   See “Program Binding” on page 658 for guidelines on the specific structure and format of the PFID and program binding.
2. On distributed directories supporting attribute types, the CPI Communications program installation object is uniquely identified as an object class with a registered ISO object identifier of 1.3.18.0.2.6.7.

Using the Distributed Directory

Figure 4 on page 27 illustrates two ways that a CPI Communications program might use destination information stored in a distributed directory:

- The program accesses the distributed directory directly with a DN to retrieve a program installation object (**1** and **2**). The program installation object contains a program binding as one of its pieces of information. The program passes the program binding to CPI Communications (**3**) using the Set_Partner_ID call with the *partner_ID_type* parameter set to CM_PROGRAM_BINDING. CPI Communications then uses the program-binding information to allocate the conversation.
- Instead of accessing the directory itself, the program passes a DN for a program installation object to CPI Communications (**3**) using the Set_Partner_ID call with a *partner_ID_type* parameter set to CM_DISTINGUISHED_NAME. CPI Communications uses the DN to access the

distributed directory and retrieve the program installation object (4). CPI Communications then uses the program-binding information from the object to allocate the conversation.

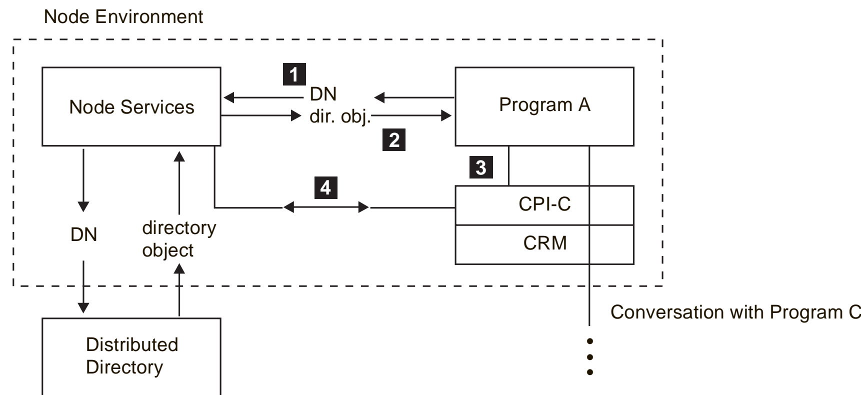


Figure 4. Program Interaction with CPI Communications and a Distributed Directory

Both these examples require the program to provide a DN directly, either to node services or to CPI Communications. There are, however, several other ways a program can access information contained in the distributed directory:

- The program provides a *sym_dest_name* on the Initialize_Conversation call that corresponds to a side information entry containing a DN. CPI Communications will then use the DN to access the distributed directory and retrieve the program binding. After the conversation is allocated, the program can use "Extract_Partner_ID (CMEPID)" to determine the program binding used.
- The program passes a PFID to CPI Communications using "Set_Partner_ID (CMSPID)" with a *partner_ID_type* parameter set to CM_PROGRAM_FUNCTION_ID. CPI Communications uses the PFID to search the distributed directory and retrieve a program installation object providing the appropriate function. CPI Communications then uses the program-binding information from the object to allocate the conversation.
- The program accesses the distributed directory using node services and locates the appropriate directory object (and program binding) using something other than a DN. For example, the program might search the directory for non-CPI-C information to determine the correct program installation object. Once located, the program binding from the directory object can be passed to CPI Communications using "Set_Partner_ID (CMSPID)" with a *partner_ID_type* parameter set to CM_PROGRAM_BINDING.

Interaction with Side Information and Set Calls

CPI Communications does not attempt to integrate destination information from the distributed directory with destination information obtained from Set calls or side information. The following rules apply:

- If the *partner_ID* characteristic is null, CPI Communications will use the values of the destination information obtained from Set calls or side information.
- If a *partner_ID* is non-null, Set calls to provide alternative destination information (other than Set_Partner_ID, Set_Conversation_Security_Type, Set_Security_User_ID, and Set_Security_Password) are not allowed. The program is notified of the error condition with a *return_code* of CM_PROGRAM_PARAMETER_CHECK.

- If a non-null *partner_ID* is available, CPI Communications will use it to establish the conversation. CPI Communications will ignore any other destination information (other than security information) that may have been established by Set calls or side information.

Distributed Security

Distributed security allows a user or system to be defined at a single trusted authentication server using **principal names** rather than user IDs. For example, a user or program accessing three different systems might require three separate sets of user IDs and passwords, one for each system. Using distributed security, a user or system could use a single principal name and password to access all three systems.

Figure 5 shows how a distributed security service works. In this example, the user has already signed on with a principal name and has been authenticated to the local security service interface. After this initial sign-on, the user is not required to provide any additional security information. When the user executes a program that requests a conversation from the CRM (**1**), the CRM communicates with the local security service interface (**2**) and retrieves the security information to be sent with the conversation startup request. The local security service communicates with the Authentication Server (**3**) to determine this information, also referred to as authentication tokens, and returns it to the CRM (**4**). The CRM then sends the security information to the partner CRM in the conversation startup request (**5**). When the partner CRM receives the conversation startup request, it accesses its local security service interface and validates the authentication tokens (**6** and **7**).

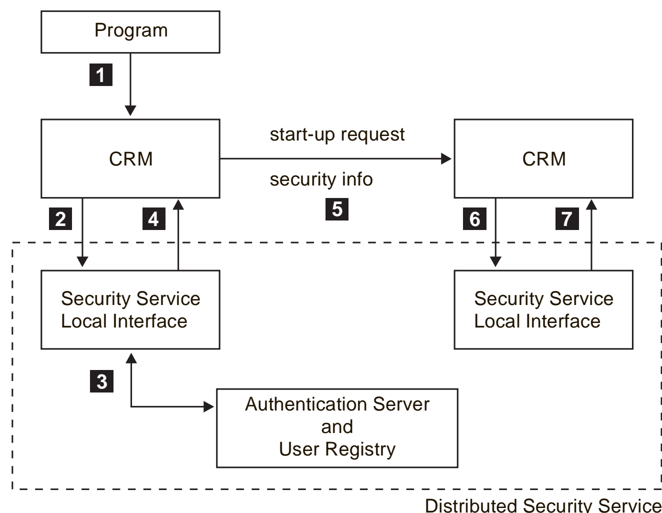


Figure 5. CRM Interaction with Distributed Security Service

Operating System

CPI Communications depends on the operating system for the normal execution and operation of the program. Activities such as linking, invoking, and compiling programs are all described in product documentation. See the product chapters, “Part 3. CPI-C 2.1 Implementation Specifics” on page 373, for more specific information.

Program Calls

CPI Communications programs communicate with each other by making program **calls.** These calls are used to establish the **characteristics** of the conversation and to exchange data and control information between the programs. An example of a conversation characteristic is the *send_receive_mode* characteristic, which indicates whether or not both programs may send data at the same time. Conversation characteristics are discussed in greater detail in "Conversation Characteristics" on page 33.

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

The *return_code* output parameter is returned for all CPI Communications calls. It indicates whether a call completed successfully or if an error was detected that caused the call to fail. CPI Communications uses additional output parameters on some calls to pass status information to the program. These parameters include the *control_information_received*, *data_received*, and *status_received* parameters. Additionally, the return code may be associated with **secondary information**, which can be used to determine the cause of the return code.

Starter Set Calls

The following six calls are called the starter set.

Initialize_Conversation
Accept_Conversation
Allocate
Send_Data
Receive
Deallocate

These six calls, which provide the core function, allow for simple communication of data between two programs. They are sufficient for writing simple applications that use the initial values for the CPI-C conversation characteristics. They are necessary for writing very complex applications. Example flows using these calls are provided in "Starter-Set Flows" on page 68.

Establishing a Conversation

Here is a simple example of how Program A starts a conversation with Program C:

1. Program A issues the Initialize_Conversation call to prepare to start the conversation. It uses a *sym_dest_name* to designate Program C as its partner program. The CRM returns a unique conversation identifier, the *conversation_ID*. Program A will use this *conversation_ID* in all future calls intended for that conversation.
2. Program A issues an Allocate call to start the conversation.
3. CPI Communications tells the node containing Program C that Program C needs to be started by sending a **conversation startup request** (in LU 6.2, this is an attach) to the partner CRM. The conversation startup request

contains information necessary to start the partner program and establish the conversation.

4. Program C is started and issues the Accept_Conversation call. It receives back a unique *conversation_ID* (not necessarily the same as the one provided to Program A). Program C will use its *conversation_ID* in all future calls intended for that conversation.

After issuing their respective Initialize_Conversation and Accept_Conversation calls, both Program A and Program C have a set of default **conversation characteristics** set up for the conversation. The default values established by CPI Communications are discussed in "Conversation Characteristics" on page 33.

Multiple Conversations

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

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

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

Partner Program Names

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

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

- Receipt of a conversation startup request
- Local action

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

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

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

Multiple Outbound Conversations

Figure 6 on page 31 shows Program A establishing conversations with two partners. For example, a program may need to request data from multiple data bases on different nodes to answer a particular query. The conversation with Program B is initialized with an Initialize_Conversation (CMINIT) call that returns a *conversation_ID* parameter of X. The conversation with Program C is initialized with an Initialize_Conversation call that returns a *conversation_ID* parameter of Y.

When Program A issues subsequent calls with a *conversation_ID* of X, CPI Communications will know these calls apply to the conversation with Program B. Similarly, when Program A issues subsequent calls with a *conversation_ID* of Y, CPI Communications will know these calls apply to the conversation with Program C.

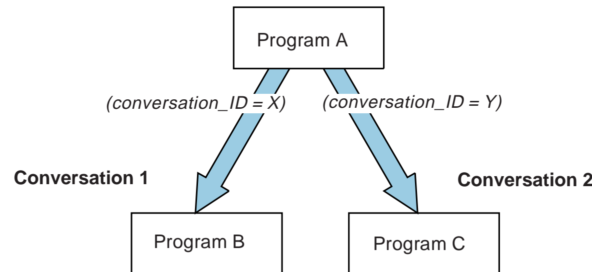


Figure 6. A Program Using Multiple Outbound CPI Communications Conversations

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

Multiple Inbound Conversations

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

This type of application is shown in Figure 7, where Programs D and E have both chosen to initiate conversations with the same partner, Program S.

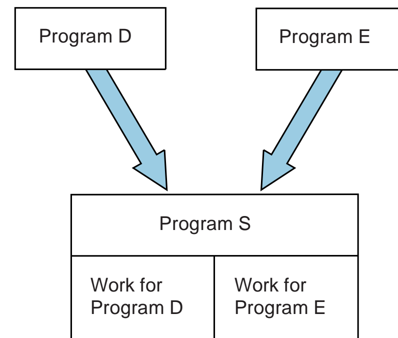


Figure 7. A Program Using Multiple Inbound CPI Communications Conversations

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

Note: A program would use the advanced calls to achieve greater programming flexibility. See "Non-Blocking Operations" on page 47 for a more detailed discussion. See "Example 12: Accepting Multiple Conversations Using Blocking Calls" on page 92 and "Example 13: Accepting Multiple Conversations Using Conversation-Level Non-Blocking Calls" on page 94 for examples using these calls.

Contexts and Context Management

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

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

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

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

Relationship between Contexts and Conversations

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

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

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

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

Relationship between Contexts and Security Parameters

Security parameters are among the logical attributes maintained by node services for a context. The access security information carried in the conversation startup request is used to set the security parameters for the context created as a result of an incoming conversation. The security parameters for other contexts are based on security information maintained within the system. See “Conversation Security” on page 51 for a complete discussion of conversation security.

Inbound and Outbound Conversations

A program that accepts incoming conversations from multiple program partners must ensure that work is done within the right context. In the expanded server example shown in Figure 8, Program S accepts two conversations, one each from Programs D and E. Two new contexts are created, one for the work done on behalf of Program D and one for the work done on behalf of Program E. Program S can retrieve the context identifier for each conversation by issuing the Extract_Conversation_Context call twice.

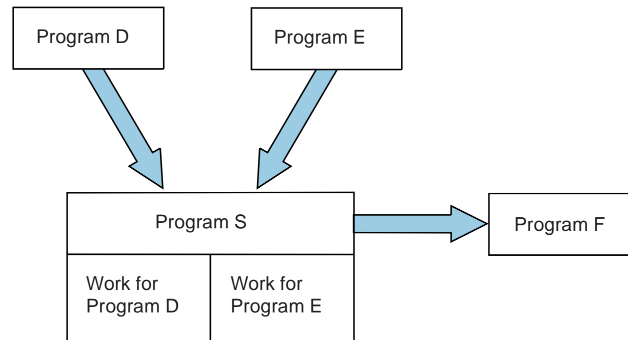


Figure 8. Server Program with Both Inbound and Outbound Conversations

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

Before allocating the conversation to Program F, Program S must first ensure that it is using the correct context. It does this by setting the current context to that for Program D. This causes the outgoing conversation to be established using information, such as the proper security parameters, from the correct context. "Conversation Security" on page 51 provides a complete discussion of how the current context is used by node services to establish security parameters for new conversations.

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

- "Non-Blocking Operations" on page 47 for use of non-blocking calls
- "Example 1: Data Flow in One Direction" on page 69
- "Example 12: Accepting Multiple Conversations Using Blocking Calls" on page 92
- "Example 13: Accepting Multiple Conversations Using Conversation-Level Non-Blocking Calls" on page 94
- Usage notes for "Accept_Conversation (CMACCP)" on page 119
- Usage notes for "Accept_Incoming (CMACCI)" on page 121
- Usage notes for "Initialize_Conversation (CMINIT)" on page 200
- Usage notes for "Initialize_For_Incoming (CMINIC)" on page 203

Conversation Characteristics

CPI Communications maintains a set of characteristics for each conversation used by a program. These characteristics are established for each program on a per-conversation basis, and the initial values assigned to the characteristics depend on the program's role in starting the conversation. Table 3 on page 35 provides a comparison of the conversation characteristics and initial values as set by the Initialize_Conversation, Accept_Conversation, Initialize_For_Incoming, and Accept_Incoming calls. The uppercase values shown in the table are pseudonyms that represent integer values.

The CPI Communications naming conventions for these characteristics, as well as for calls, variables, and characteristic values, are discussed in "Naming Conventions—Calls, Characteristics, Variables, and Values" on page 13.

Modifying and Viewing Characteristics

In the example in "Establishing a Conversation" on page 29, the programs used the initial set of program characteristics provided by CPI Communications as defaults. However, CPI Communications provides calls that allow a program to modify and view the conversation characteristics for a particular conversation. Restrictions on when a program can issue one of these calls are discussed in the individual call descriptions in Chapter 4, "Call Reference."

Note: CPI Communications maintains conversation characteristics on a per-conversation basis. Changes to a characteristic will affect only the conversation indicated by the *conversation_ID*. Changes made to a characteristic do not affect future default values assigned, nor do the changes affect the initial system values (in the case of values derived from the side information).

For example, consider the conversation characteristic that defines what type of conversation the initiating program will have, the *conversation_type* characteristic. CPI Communications initially sets this characteristic to CM_MAPPED_CONVERSATION and stores this characteristic value for use in maintaining the conversation. A program can issue the Extract_Conversation_Type call to view this value.

A program can issue the Set_Conversation_Type call (after issuing Initialize_Conversation but before issuing Allocate) to change this value. The change remains in effect until the conversation ends or until the program issues another Set_Conversation_Type call.

The Set calls are also used to prevent programs from attempting incorrect syntactic or semantic changes to conversation characteristics. For example, if a program attempts to change the *conversation_type* after the conversation has already been established (an illegal change), CPI Communications informs the program of its error and disallows the change. Details on this type of checking are provided in the individual call descriptions in Chapter 4, "Call Reference."

Table 3 (Page 1 of 4). Characteristics and Their Default Values

Name of Characteristic	Initialize_Conversation sets it to:	Accept_Conversation sets it to:	Initialize_For_Incoming sets it to:	Accept_Incoming sets it to:
AE_qualifier	The application-entity qualifier from side information referenced by *sym_dest_name*. If a blank *sym_dest_name* was specified, *AE_qualifier* will be the null string.	For an OSI TP CRM, the initiating *AE_qualifier* received on the conversation startup request. For an LU 6.2 CRM, the null string.	Not set	For an OSI TP CRM, the initiating *AE_qualifier* received on the conversation startup request. For an LU 6.2 CRM, the null string.
AE_qualifier_length	The length of *AE_qualifier*. If a blank *sym_dest_name* was specified, *AE_qualifier_length* will be 0.	The length of *AE_qualifier*	Not set	The length of *AE_qualifier*
AE_qualifier_format	The format of *AE_qualifier*. If a blank *sym_dest_name* was specified, *AE_qualifier_format* will not be meaningful.	For an OSI TP CRM, the format of *AE_qualifier*. For an LU 6.2 CRM, *AE_qualifier_format* is not set.	Not set	For an OSI TP CRM, the format of *AE_qualifier*. For an LU 6.2 CRM, *AE_qualifier_format* is not set.
allocate_confirm	CM_ALLOCATE_NO_CONFIRM	Not applicable	Not applicable	Not applicable
AP_title	The application-process title from side information referenced by *sym_dest_name*. If a blank *sym_dest_name* was specified, *AP_title* will be the null string.	For an OSI TP CRM, the initiating *AP_title* received on the conversation startup request. For an LU 6.2 CRM, the null string.	Not set	For an OSI TP CRM, the initiating *AP_title* received on the conversation startup request. For an LU 6.2 CRM, the null string.
AP_title_length	The length of *AP_title*. If a blank *sym_dest_name* was specified, *AP_title_length* will be 0.	The length of *AP_title*	Not set	The length of *AP_title*
AP_title_format	The format of *AP_title*. If a blank *sym_dest_name* was specified, *AP_title_format* will not be meaningful.	For an OSI TP CRM, the format of *AP_title*. For an LU 6.2 CRM, *AP_title_format_* is not set.	Not set	For an OSI TP CRM, the format of *AP_title*. For an LU 6.2 CRM, *AP_title_format_* is not set.
application_context_name	The application context name from side information referenced by *sym_dest_name*. If a blank *sym_dest_name* was specified, *application_context_name* will be the null string.	For an OSI TP CRM, the initiating *application_context_name* received on the conversation startup request. For an LU 6.2 CRM, the null string.	Not set	For an OSI TP CRM, the initiating *application_context_name* received on the conversation startup request. For an LU 6.2 CRM, the null string.
application_context_name_length	The length of *application_context_name*. If a blank *sym_dest_name* was specified, *application_context_name* will be 0.	The length of *application_context_name*	Not set	The length of *application_context_name*
begin_transaction	CM_BEGIN_IMPLICIT	Not applicable	Not applicable	Not applicable
confirmation_urgency	CM_CONFIRMATION_URGENT	CM_CONFIRMATION_URGENT	CM_CONFIRMATION_URGENT	Not changed by Accept_Incoming
context_ID	Not set	The *context_ID* of the newly created context	Not set	The *context_ID* of the newly created context
context_ID_length	Not set	The length of *context_ID*	Not set	The length of *context_ID*

Table 3 (Page 2 of 4). Characteristics and Their Default Values

Name of Characteristic	Initialize_Conversation sets it to:	Accept_Conversation sets it to:	Initialize_For_Incoming sets it to:	Accept_Incoming sets it to:
conversation_security_type	The security type from side information referenced by *sym_dest_name*. If a blank *sym_dest_name* was specified, *conversation_security_type* will be CM_SECURITY_SAME.	Not applicable	Not applicable	Not applicable
conversation_state	CM_INITIALIZE_STATE	For half-duplex conversations, CM_RECEIVE_STATE. For full-duplex conversations, CM_SEND_RECEIVE_STATE.	CM_INITIALIZE_INCOMING_STATE	For half-duplex conversations, CM_RECEIVE_STATE. For full-duplex conversations, CM_SEND_RECEIVE_STATE.
conversation_type	CM_MAPPED_CONVERSATION	CM_MAPPED_CONVERSATION if the CRM type is OSI TP, or the value received on the conversation startup request if the CRM type is LU 6.2.	CM_MAPPED_CONVERSATION	The value received on the conversation startup request if the CRM type is LU 6.2. Not changed if the CRM type is OSI TP.
deallocate_type	CM_DEALLOCATE_SYNC_LEVEL	CM_DEALLOCATE_SYNC_LEVEL	CM_DEALLOCATE_SYNC_LEVEL	Not changed by Accept_Incoming
directory_encoding	The *directory_encoding* referenced by *sym_dest_name*. If a blank *sym_dest_name* was specified or the side information entry did not contain a *directory_encoding* value, *directory_encoding* will be set to CM_DEFAULT_ENCODING.	Not applicable	Not set	Not applicable
directory_syntax	The *directory_syntax* referenced by *sym_dest_name*. If a blank *sym_dest_name* was specified or the side information entry did not contain a *directory_syntax* value, *directory_syntax* will be set to CM_DEFAULT_SYNTAX.	Not applicable	Not set	Not applicable
error_direction	CM_RECEIVE_ERROR	CM_RECEIVE_ERROR	CM_RECEIVE_ERROR	Not changed by Accept_Incoming
fill	CM_FILL_LL	CM_FILL_LL	CM_FILL_LL	Not changed by Accept_Incoming
initialization_data	Null	The value received on the conversation startup request	Null	The value received on the conversation startup request
initialization_data_length	0	The length of the initialization data received on the conversation startup request	0	The length of the initialization data received on the conversation startup request
log_data	Null	Null	Null	Not changed by Accept_Incoming
join_transaction	Not set	CM_JOIN_IMPLICIT	CM_JOIN_IMPLICIT	Not changed by Accept_Incoming
log_data_length	0	0	0	Not changed by Accept_Incoming
map_name	Null	Null	Null	Not changed by Accept_Incoming
map_name_length	0	0	0	Not changed by Accept_Incoming

Table 3 (Page 3 of 4). Characteristics and Their Default Values

Name of Characteristic	Initialize_Conversation sets it to:	Accept_Conversation sets it to:	Initialize_For_Incoming sets it to:	Accept_Incoming sets it to:
mode_name	The mode name from side information referenced by *sym_dest_name*. If a blank *sym_dest_name* was specified, *mode_name* will be the null string.	The mode name for the logical connection on which the conversation startup request arrived	Not set	The mode name for the logical connection on which the conversation startup request arrived
mode_name_length	The length of *mode_name*. If a blank *sym_dest_name* was specified, *mode_name_length* will be 0.	The length of *mode_name*	Not set	The length of *mode_name*
partner_ID	The *distinguished_name* referenced by *sym_dest_name*. If a blank *sym_dest_name* was specified or the side information entry did not contain a *distinguished_name*, *partner_ID* will be the null string.	A program binding containing all available destination information on the partner program.	Not set	A program binding containing all available destination information on the partner program.
partner_ID_length	The length of *partner_ID*. If *partner_ID* is null, *partner_ID_length* will be 0.	The length of *partner_ID*.	Not set	The length of *partner_ID*.
partner_ID_type	CM_DISTINGUISHED_NAME	CM_PROGRAM_BINDING	Not set	CM_PROGRAM_BINDING
partner_ID_scope	CM_EXPLICIT	Not applicable	Not set	Not applicable
partner_LU_name	The partner LU name from side information referenced by *sym_dest_name*. If a blank *sym_dest_name* was specified, *partner_LU_name* will be a single blank.	For an LU 6.2 CRM, the partner LU name for the logical connection on which the conversation startup request arrived. For an OSI TP CRM, *partner_LU_name* is a single blank.	Not set	For an LU 6.2 CRM, the partner LU name for the logical connection on which the conversation startup request arrived. For an OSI TP CRM, *partner_LU_name* is a single blank.
partner_LU_name_length	The length of *partner_LU_name*. If a blank *sym_dest_name* was specified, *partner_LU_name_length* will be 1.	The length of *partner_LU_name*	Not set	The length of *partner_LU_name*
prepare_data_permitted	CM_PREPARE_DATA_NOT_PERMITTED	Not applicable	Not applicable	Not applicable
prepare_to_receive_type	CM_PREP_TO_RECEIVE_SYNC_LEVEL	CM_PREP_TO_RECEIVE_SYNC_LEVEL	CM_PREP_TO_RECEIVE_SYNC_LEVEL	Not changed by Accept_Incoming
processing_mode	CM_BLOCKING	CM_BLOCKING	CM_BLOCKING	Not changed by Accept_Incoming
receive_type	CM_RECEIVE_AND_WAIT	CM_RECEIVE_AND_WAIT	CM_RECEIVE_AND_WAIT	Not changed by Accept_Incoming
return_control	CM_WHEN_SESSION_ALLOCATED	Not applicable	Not applicable	Not applicable
security_password	The security password from side information referenced by *sym_dest_name*. If a blank *sym_dest_name* was specified, *security_password* will be the null string.	Not applicable	Not applicable	Not applicable
security_password_length	The length of *security_password*. If a blank *sym_dest_name* was specified, *security_password_length* will be 0.	Not applicable	Not applicable	Not applicable

Table 3 (Page 4 of 4). Characteristics and Their Default Values

Name of Characteristic	Initialize_Conversation sets it to:	Accept_Conversation sets it to:	Initialize_For_Incoming sets it to:	Accept_Incoming sets it to:
security_user_ID	The security user ID from side information referenced by *sym_dest_name*. If a blank *sym_dest_name* was specified, *security_user_ID* will be the null string.	The value received on the conversation startup request	Not set	The value received on the conversation startup request
security_user_ID_length	The length of *security_user_ID*. If a blank *sym_dest_name* was specified, *security_user_ID_length* will be 0.	The length of *security_user_ID*	Not set	The length of *security_user_ID*
send_receive_mode	CM_HALF_DUPLEX	The value received in the conversation startup request	Not set	The value received in the conversation startup request
send_type	CM_BUFFER_DATA	CM_BUFFER_DATA	CM_BUFFER_DATA	Not changed by Accept_Incoming
sync_level	CM_NONE	The value received on the conversation startup request	Not set	The value received on the conversation startup request
TP_name	The program name from side information referenced by *sym_dest_name*. If a blank *sym_dest_name* was specified, *TP_name* will be a single blank.	The value received on the conversation startup request	Not set	The value received on the conversation startup request
TP_name_length	The length of *TP_name*. If a blank *sym_dest_name* was specified, *TP_name_length* will be 1.	The length of *TP_name*	Not set	The length of *TP_name*
transaction_control	CM_CHAINED_TRANSACTIONS	For an OSI TP CRM, the value received on the conversation startup request. For an LU 6.2 CRM, CM_CHAINED_TRANSACTIONS.	Not set	For an OSI TP CRM, the value received on the conversation startup request. For an LU 6.2 CRM, CM_CHAINED_TRANSACTIONS.

Characteristic Values and CRMs

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

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

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

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

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

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

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

  Details on this type of checking are provided in the individual call descriptions in Chapter 4, “Call Reference” on page 107.

- The conversation will only be allocated on that CRM type.

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

Characteristic Values and Send-Receive Modes

Table 4 lists the values of conversation characteristics that are not applicable to a **half-duplex conversation**, a conversation with *send_receive_mode* set to CM_HALF_DUPLEX. Table 5 lists the values of conversation characteristics that are not applicable to a **full-duplex conversation**, a conversation with *send_receive_mode* set to CM_FULL_DUPLEX.

Table 4. Conversation Characteristic Values that Cannot Be Set for Half-Duplex Conversations

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

Table 5. Conversation Characteristic Values that Cannot Be Set for Full-Duplex Conversations

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

On a conversation with a particular send-receive mode:

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

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

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

Automatic Conversion of Characteristics

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

When an LU 6.2 CRM is used, CPI Communications requires that these transmitted characteristics be encoded as EBCDIC characters. For this reason, CPI Communications automatically converts these characteristics to EBCDIC when they are used as parameters on CPI Communications calls on non-EBCDIC systems. The conversion of characteristics not in character set 00640 is implementation dependent (see Table 60 on page 647). When an OSI TP CRM is used, the transfer syntax is negotiated by the underlying support. CPI Communications automatically converts these characteristics to the transfer syntax when they are used as parameters on CPI Communications calls.

This means programmers can use the native encoding of the local system when specifying these characteristics on Set calls. Likewise, when these characteristics are returned by Extract calls, they are represented in the local system's native encoding.

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

AE_qualifier — Specified on the Extract_AE_Qualifier and Set_AE_Qualifier calls.

AP_Title — Specified on the Extract_AP_Title and Set_AP_Title calls.

application_context_name — Specified on the Extract_Application_Context_Name and Set_Application_Context_Name calls.

initialization_data — Specified on the Extract_Initialization_Data and Set_Initialization_Data calls.

log_data — Specified on the Set_Log_Data call.

mode_name — Specified on the Extract_Mode_Name and Set_Mode_Name calls.

partner_LU_name — Specified on the Extract_Partner_LU_Name and Set_Partner_LU_Name calls.

security_password — Specified on the Set_Conversation_Security_Password call.

security_user_ID — Specified on the Extract_Security_User_ID and Set_Conversation_Security_User_ID calls.

TP_name — Specified on the Set_TP_Name and Extract_TP_Name calls. Refer to “SNA Service Transaction Programs” on page 727 for special handling of SNA Service Transaction Program names.

Automatic Data Conversion

Automatic data conversion allows two mapped conversation programs to exchange data records in a manner that insulates a program from concerns about how the partner program views the data record. This is accomplished by encoding and decoding data records using routines that are external to the user application. These routines can be written by a systems administrator, some other qualified group, or a vendor. The MAP_NAME is used to access the correct routine. Within the SNA architecture this would be a separate routine which the underlying layers would access. A particular implementation may either have the user link edit the routine into the application program or use dynamic link libraries. The results are the same to the end user. Within the OSI architecture this name would instruct the U-ASE (the OSI TP entity responsible for encoding and decoding user data) what specific U-ASE ASN.1 structure to use. When a data record is received the reverse process occurs with the decoder recognizing what it received and generating the appropriate local MAP_NAME.

When an application program is moving data between homogeneous systems and intends to perform all manipulation of the data within the application program itself, it may use the network like a pipe and simply send and receive buffers of data. The program can also accomplish this by using pure Send_Data and Receive calls. It can also accomplish this by using Send_Mapped_data and Receive_Mapped_Data calls and not supplying a map name or supplying one with all blanks. The SNA routine the vendor installed routine which supports CPI-C 1.0 mapped conversations. The OSI routine is the Unstructured Data Transfer U-ASE (UDT).

The MAP_NAME is a 1 to 64 character local name defined by the person who wrote the encode/decode routine. Both sides of the conversation could have different local names for the same mapping function. This might occur when a transaction traveled across an organizational boundary where the naming could not be coordinated. An example would be two companies which use an industry standard data record and have independently created a MAP_NAME for the data record. SNA maps the local MAP_NAME to a globally unique network MAP_NAME. The globally unique name is provided by a systems administrator. OSI uses the application context to define, among other things, the abstract syntax's (ASN.1 encodings) the conversation can use. The MAP_NAME is a local name established by a systems administrator for the U-ASE to identify what encoding to use. Again, the two different systems may have different MAP_NAMES. Unlike SNA, OSI does not transmit a MAP_NAME across the network. Instead the self defining nature of ASN.1 automatically performs this function. The encode/decode routines need not be mirror images of each other. The ability to manipulate the user data before it is sent or received by the application program allows the application programmer insulation from the network. The receiver may only require a subset of the data that was sent, therefore only specific fields are delivered to the program. This allows changes in the data a program sends to occur without requiring every receiving program to change, only the decoding routines need change. The sending program may send a subset of the required data with the encoding routine filling in the missing fields.

Automatic data conversion assumes that both partners operate correctly. There may be cases where Send_Data and Receive calls are mixed with Send_Mapped_Data and Receive_Mapped_Data. These calls can be mixed if the following assumptions are met:

1. Both partners use equivalent map routines.
2. The Send_Data partner's map routine can recognize the incoming buffer and encode correctly. For LU 6.2, the map routine will generate the correct map name and send it to the partner.
3. The Receive_Data partner application code is able to recognize the decoded buffer and act accordingly.

The send/receive mapped data routines all use a Map_Name_Length of 0 to 64. A zero length indicates a NULL map name and the use of either the LU 6.2 supplied map routine or the OSI Unstructed Data ASE, depending on the underlying CRM.

Data Conversion

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

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

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

These two calls provide limited data conversion support for character data that belongs to character set 00640, as specified in Appendix A, “Variables and Characteristics.” See the usage notes under “Convert_Incoming (CMCNVI)” on page 139 and “Convert_Outgoing (CMCNVO)” on page 141.

With the Send_Mapped_Data, Receive_Mapped_Data, Set_Mapped_Initialization_Data, and Extract_Mapped_Initialization_Data calls, CPI-C provides access to sophisticated routines for conversion of data in the user buffer before it is sent or received. These routines may be used to write programs that are independent of encoding of the partner program. This allows the encode/decode routine to change the order of the fields, selectively receive fields, and supply missing fields when sending.

These calls should not be used when using the OSI Unstructured Data Transfer U-ASE or CPI-C 1.0 mapped conversations. If you desire to use these calls for uniformity with other programs the MAP_NAME should be set to blanks and/or MAP_NAME_LENGTH set to zero.

Data Buffering and Transmission

If a program uses the initial set of conversation characteristics, data is not automatically sent to the remote program after a Send_Data has been issued, except when the send buffer at the local system overflows. As shown in the starter-set flows, the startup of the conversation and subsequent data flow can occur anytime after the program call to Allocate. This is because the system stores the data in internal buffers and groups transmissions together for efficiency.

A program can exercise explicit control over the transmission of data by using one of the following calls to cause the buffered data's immediate transmission:

- Confirm
- Deallocate
- Deferred_Deallocate
- Flush
- Include_Partner_In_Transaction
- Prepare
- Prepare_To_Receive
- Receive (in **Send** state) with *receive_type* set to CM_RECEIVE_AND_WAIT (*receive_type*'s default setting)
- Send_Error

The use of Receive in **Send** state and the use of Deallocate have already been shown in "Starter-Set Flows" on page 68. The other calls are discussed in the following examples.

Concurrent Operations

CPI Communications provides for concurrent call operations (multiple call operations in progress simultaneously) on a conversation by grouping calls in logical associations or **conversation queues**. Calls associated with one queue are processed independently of calls associated with other queues or with no queue. Table 6 on page 45 shows the different conversation queues and calls associated with them.

The send-receive mode of the conversation determines what queues are available for the conversation. Table 6 on page 45 shows the send-receive modes for which the conversation queues are available.

A program may initiate concurrent operations by using multiple program threads on systems with multi-threading support. See "Using Multiple Program Threads" on page 45. Alternatively, a program may use queue-level non-blocking support to regain control when a call operation on a queue cannot complete immediately. The call operation remains in progress. The program may issue a call associated with another queue or perform other processing. Queue-level non-blocking is described in "Queue-Level Non-Blocking" on page 49.

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

Using Multiple Program Threads

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

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

Table 6 (Page 1 of 2). Conversation Queues—Associated Calls and Send-Receive Modes

Conversation Queue	CPI Communications Calls	Send-Receive Mode
Initialization	Accept_Incoming Allocate Extract_Mapped_Initialization_Data Set_AE_Qualifier Set_Allocate_Confirm Set_AP_Title Set_Application_Context_Name Set_Conversation_Security_Password Set_Conversation_Security_Type Set_Conversation_Security_User_ID Set_Conversation_Type Set_Initialization_Data Set_Mapped_Initialization_Data Set_Mode_Name Set_Partner_ID Set_Partner_LU_Name Set_Return_Control Set_Send_Receive_Mode Set_Sync_Level Set_Transaction_Control Set_TP_Name	Half-duplex and full-duplex
Send	Confirmed Deallocate Deferred_Deallocate Flush Include_Partner_In_Transaction Prepare Send_Data Send_Error Send_Mapped_Data Set_Deallocate_Type Set_Log_Data Set_Prepare_Data_Permitted Set_Send_Type	Full-duplex
Receive	Receive Receive_Mapped_Data Set_Fill Set_Receive_Type	Full-duplex

Table 6 (Page 2 of 2). Conversation Queues—Associated Calls and Send-Receive Modes

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

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

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

Non-Blocking Operations

CPI Communications supports two processing modes for its calls:

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

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

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

The following calls can return the CM_OPERATION_INCOMPLETE return code:

Table 7. Calls Returning CM_OPERATION_INCOMPLETE

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

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

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

Conversation-Level Non-Blocking

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

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

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

issued on the conversation while the previous call operation is still in progress gets the CM_OPERATION_NOT_ACCEPTED return code.

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

Queue-Level Non-Blocking

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

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

Working with Wait Facility

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

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

Wait Facility Scenario

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

1. The program uses the Set_Queue_Processing_Mode call to set the processing mode for the Send queue to non-blocking. It also supplies a user field that contains the address of a parameter list for the Send_Data call and receives back an OOID from CPI Communications that is unique to the Send queue.
2. The program next uses the Set_Queue_Processing_Mode call to set the processing mode for the Receive queue to non-blocking. This time it supplies a user field that contains the address of a parameter list for the Receive call. It

receives back an OOID from CPI Communications that is unique to the Receive queue.

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

Using Callback Function

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

Canceling Outstanding Operations

A program may use the Cancel_Conversation (CMCANC) call to end a conversation. The call terminates all the call operations in progress on the conversation. The terminated call operation returns a code of CM_CONVERSATION_CANCELLED

Non-Blocking Calls and Context Management

In general, the program's current context is set by node services to the newly created context when an Accept_Conversation or Accept_Incoming call completes successfully with a return code of CM_OK. However, if Accept_Incoming is issued as a non-blocking call and returns with a CM_OPERATION_INCOMPLETE return code, the program's current context will not be changed. A new context is not created until the Accept_Incoming call operation subsequently completes successfully as a result of the Wait_For_Conversation or Wait_For_Completion call. The program can then use the Extract_Conversation_Context call to determine the context to which the conversation was assigned. Because Wait_For_Conversation and Wait_For_Completion do not cause a change of context, the program is responsible for issuing the appropriate node services call to establish the correct current context.

Conversation Security

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

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

- No access security information
- The user ID of the user on whose behalf access to the remote program is requested
- The user ID and a password for the user on whose behalf access to the remote program is requested
- Authentication tokens generated by a distributed security service for the user on whose behalf access to the remote program is requested.

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

- The system administrator can provide security parameters in the side information. These are used to establish security characteristics when the program issues the Initialize_Conversation call.
- The program can override the values from side information and set the security characteristics directly using the Set_Conversation_Security_Type, Set_Conversation_Security_User_ID, and Set_Conversation_Security_Password calls.
- When the program allocates a conversation with *conversation_security_type* set to CM_SECURITY_SAME, CM_SECURITY_DISTRIBUTED or CM_SECURITY_MUTUAL, the security parameters for the program's current context are used to generate the access security information. This may involve the use of a distributed security service. The access security information is sent to the remote CRM in the conversation startup request.

The required_user_name_type field in the program binding may be used to specify the type of user name required by the remote system.

- “NONE” indicates the remote system accepts and processes requests that do not contain access security information.
- “LOCAL” indicates the remote system requires a user ID that is valid at the remote system. User IDs may be sent for CM_SECURITY_PROGRAM, CM_SECURITY_PROGRAM_STRONG, CM_SECURITY_SAME, CM_SECURITY_DISTRIBUTED and CM_SECURITY_MUTUAL requests.
- “PRINCIPAL” indicates the remote system requires a principal name from the distributed security service. Principal names are sent for CM_SECURITY_SAME, CM_SECURITY_DISTRIBUTED and CM_SECURITY_MUTUAL requests.

Certain combinations of values of *conversation_security_type* and required_user_name_type field (from the program binding) cause the local CRM to

reject the Allocate call with a *return_code* of CM_SECURITY_NOT_SUPPORTED. Table 8 on page 52 shows the incompatible values.

Table 8. Incompatible conversation_security_type and required_user_name_type Values

required_user_name_type	conversation_security_type
LOCAL	CM_SECURITY_NONE
PRINCIPAL	CM_SECURITY_NONE
PRINCIPAL	CM_SECURITY_PROGRAM
PRINCIPAL	CM_SECURITY_PROGRAM_STRONG

In addition to supporting access security information on conversation startup requests, the OSI standard includes the ability to perform re-authentication during a conversation. A program requests re-authentication by making OSI TP implementation-specific calls to node services. Security protocols for OSI TP CRMs are defined in standard ISO/IEC 11586 part 1, *Generic Upper Layers Security*.

For an OSI TP CRM, the application context identifies to the program developer which conversation security type should be used.

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

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

Program Flow—States and Transitions

As implied throughout the discussion so far, a program written to make use of CPI Communications is written with the remote program in mind. The local program issues a CPI Communications call for a particular conversation knowing that, in response, the remote program will issue another CPI Communications call (or its equivalent) for that same conversation. To explain this two-sided programming scenario, CPI Communications uses the concept of a conversation state. The **state** that a conversation is in determines what the next set of actions may be. When a conversation leaves a state, it makes a **transition** from that state to another.

A CPI Communications conversation can be in one of the following states:

State	Description
Reset	There is no conversation for this *conversation_ID*.
Initialize	Initialize_Conversation has completed successfully and a *conversation_ID* has been assigned for this conversation.
Send	The program is able to send data on this conversation. This state is applicable only for half-duplex conversations.
Receive	The program is able to receive data on this conversation. This state is applicable only for half-duplex conversations.

State	Description
Send-Pending	The program has received both data and send control on the same Receive call. See “Example 7: Error Direction and Send-Pending State” on page 82 for a discussion of the **Send-Pending** state. This state is applicable only for half-duplex conversations.
Confirm	A confirmation request has been received on this conversation; that is, the remote program issued either a Confirm call or a Send_Data call with *send_type* set to CM_SEND_AND_CONFIRM, and is waiting for the local program to issue Confirmed. After responding with Confirmed, the local program's end of the conversation enters **Receive** state. This state is applicable only for half-duplex conversations.
Confirm-Send	A confirmation request and send control have both been received on this conversation; that is, the remote program issued a Prepare_To_Receive call with the *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and either the *sync_level* is CM_CONFIRM, or the *sync_level* is CM_SYNC_POINT and the conversation is not currently included in a transaction. After responding with Confirmed, the local program's end of the conversation enters **Send** state. This state is applicable only for half-duplex conversations.
Confirm-Deallocate	A confirmation request and deallocation notification have both been received on this conversation. For a half-duplex conversation, the remote program issued a Deallocate call in one of the following situations: • *deallocate_type* is set to CM_DEALLOCATE_CONFIRM. • *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is set to CM_CONFIRM. • *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT, and the conversation is not currently included in a transaction. For a full-duplex conversation, the remote program issued a Deallocate call with the *deallocate_type* set to CM_DEALLOCATE_CONFIRM.
Initialize-Incoming	Initialize_For_Incoming has completed successfully and a *conversation_ID* has been assigned for this conversation. The program may accept an incoming conversation by issuing Accept_Incoming on this conversation.
Send-Receive	The program can send and receive data on this conversation. This state is applicable only for full-duplex conversations.
Send-Only	The program can only send data on this conversation. This state is applicable only for full-duplex conversations.
Receive-Only	The program can only receive data on this conversation. This state is applicable only for full-duplex conversations.

A conversation starts out in **Reset** state and moves into other states, depending on the calls made by the program for that conversation and the information received from the remote program. The current state of a conversation determines what calls the program can or cannot make.

Since there are two programs for each conversation (one at each end), the state of the conversation *as seen by each program* may be different. The state of the conversation depends on which end of the conversation is being discussed. Consider a half-duplex conversation where Program A is sending data to Program C. Program A's end of the conversation is in **Send** state, but Program C's end is in **Receive** state.

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

The following additional states are required for programs using a *sync_level* of CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM:

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

"Support for Resource Recovery Interfaces" discusses synchronization point processing and describes these additional states.

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

Support for Resource Recovery Interfaces

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

While CPI Communications' sync point functions can be used with other resource recovery interfaces, this book describes how CPI Communications works with the Systems Application Architecture* (SAA*) CPI Resource Recovery and X/Open TX resource recovery interfaces. For information about performing synchronization point processing with the SAA resource recovery interface, see *SAA Common Programming Interface: Resource Recovery Reference* (SC31-6821) and read the documentation for the appropriate operating environment. For information on using the X/Open TX resource recovery interface, see *Distributed Transaction Processing: The TX (Transaction Demarcation) Specification*, published by X/Open Company Limited, ISBN: 1-872630-65-0.

Notes:

1. The following discussion is intended for programmers using CPI Communications advanced functions. Readers not interested in a high degree of synchronization need not read this section and can go to the next chapter.
2. The information in this section applies only to the CICS, OS/400, and VM environments, since these products support the use of the SAA resource recovery interface with CPI Communications.

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

Coordination with Resource Recovery Interfaces

A program communicates with a resource recovery interface by establishing **synchronization points**, or **sync points**, in the program logic. A sync point is a reference point during transaction processing to which resources can be restored if a failure occurs. The program uses a resource recovery interface's commit call to establish a new sync point or a resource recovery interface's backout call to return to a previous sync point. The processing and the changes to resources that occur between one sync point and the next are collectively referred to as a **transaction** or a **logical unit of work**.

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

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

For example, Program S in Figure 7 on page 31 has two distinct contexts, one for Program D and one for Program E. If Program S decides to commit or back out the work that it has done for Program D, it must set the current context to ensure that only the resources associated with Program D will be affected. For more information on contexts, see “Contexts and Context Management” on page 32.

Take-Commit and Take-Backout Notifications

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

When the program issues a Prepare or commit call, CPI Communications returns a **take-commit notification** to the partner program in the *status_received* parameter for a Receive call issued by the partner. The sequence of CPI Communications calls issued before the Prepare or resource recovery commit call determines the value of the take-commit notification returned to the partner program. In addition to

requesting that the partner program establish a sync point, the take-commit notification also contains conversation state transition information.

Table 9 and Table 10 show the *status_received* values that CPI Communications uses as take-commit notifications, the conditions under which each of the values may be received, and the state changes resulting from their receipt.

Table 9 (Page 1 of 2). Possible Take-Commit Notifications for Half-Duplex Conversations

status_received **Value**	**Conditions for Receipt**
CM_TAKE_COMMIT	The partner program issued a commit call, or a Prepare call with the *prepare_data_permitted* conversation characteristic set to CM_PREPARE_DATA_NOT_PERMITTED, while its end of the conversation was in **Send** or **Send-Pending** state. The local program's end of the conversation is in **Sync-Point** state and will be placed back in **Receive** state once the local program issues a successful commit call.
CM_TAKE_COMMIT_SEND	The partner program issued a commit call, or a Prepare call with the *prepare_data_permitted* conversation characteristic set to CM_PREPARE_DATA_NOT_PERMITTED, while its end of the conversation was in **Defer-Receive** state. The local program's end of the conversation is in **Sync-Point-Send** state and will be placed in **Send** state once the local program issues a successful commit call.
CM_TAKE_COMMIT_DEALLOCATE	The partner program issued a commit call, or a Prepare call with the *prepare_data_permitted* conversation characteristic set to CM_PREPARE_DATA_NOT_PERMITTED, either while its end of the conversation was in **Defer-Deallocate** state or after issuing a Deferred_Deallocate call. The local program's end of the conversation is in **Sync-Point-Deallocate** state and will be placed in **Reset** state once the local program issues a successful commit call.
CM_TAKE_COMMIT_DATA_OK	The partner program issued a Prepare call with the *prepare_data_permitted* conversation characteristic set to CM_PREPARE_DATA_PERMITTED while its end of the conversation was in **Send** or **Send-Pending** state. The local program's end of the conversation is in **Sync-Point** state and will be placed back in **Receive** state once the local program issues a successful commit call.

Table 9 (Page 2 of 2). Possible Take-Commit Notifications for Half-Duplex Conversations

status_received **Value**	**Conditions for Receipt**
CM_TAKE_COMMIT_SEND_DATA_OK	The partner program issued a Prepare call with the *prepare_data_permitted* conversation characteristic set to CM_PREPARE_DATA_PERMITTED while its end of the conversation was in **Defer-Receive** state. The local program's end of the conversation is in **Sync-Point-Send** state and will be placed in **Send** state once the local program issues a successful commit call.
CM_TAKE_COMMIT_DEALLOC_DATA_OK	The partner program issued a Prepare call with the *prepare_data_permitted* conversation characteristic set to CM_PREPARE_DATA_PERMITTED, either while its end of the conversation was in **Defer-Deallocate** state or after issuing a Deferred_Deallocate call. The local program's end of the conversation is in **Sync-Point-Deallocate** state and will be placed in **Reset** state once the local program issues a successful commit call.

Table 10 (Page 1 of 2). Possible Take-Commit Notifications for Full-Duplex Conversations

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

Table 10 (Page 2 of 2). Possible Take-Commit Notifications for Full-Duplex Conversations

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

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

- CM_TAKE_BACKOUT
- CM_DEALLOCATED_ABEND_BO
- CM_DEALLOCATED_ABEND_SVC_BO (basic conversations only)
- CM_DEALLOCATED_ABEND_TIMER_BO (basic conversations only)
- CM_RESOURCE_FAIL_NO_RETRY_BO
- CM_RESOURCE_FAILURE_RETRY_BO
- CM_DEALLOCATED_NORMAL_BO

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

- Confirm
- Deallocate(S)[4]
- Extract_Conversation_State
- Flush
- Prepare
- Prepare_To_Receive
- Receive
- Send_Data
- Send_Error

The Backout-Required Condition

Upon receipt of a take-backout notification on a protected conversation, the conversation's context is placed in the **Backout-Required** condition. This condition is not a conversation state, because it applies to all of the program's protected resources for that context, possibly including multiple conversations.

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

- When CPI Communications returns a take-backout notification

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

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

When a context is placed in the **Backout-Required** condition, the program should issue a resource recovery backout call. Until it issues a backout call, the program will be unable to successfully issue any of the following CPI Communications calls for any of its protected conversations within that context. The CM_PROGRAM_STATE_CHECK return code will be returned if the program issues any of the following calls:

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

Responses to Take-Commit and Take-Backout Notifications

A program usually issues a commit or backout call in response to a take-commit notification, and a backout call in response to a take-backout notification. In some cases, however, the program may respond to one of these notifications with a CPI Communications call instead of a commit or backout call. Table 11 on page 60 shows the calls a program can use to respond to take-commit and take-backout notifications, the result of issuing each call, and any further action required by the program.

Table 11. Responses to Take-Commit and Take-Backout Notifications				
Notification Received	Possible Response	Reason for Response	Result of Response	Further Action Required
Take-Commit[1]	Commit	The program agrees that it can commit (or has committed) all protected resources.	The commit request is spread to other programs in the transaction.	None
	Backout	The program disagrees with the commit request.	A backout request is spread to other programs in the transaction, including the program that issued the original commit call.	None
	Deallocate (Abend)[2] or Cancel_ Conversation	The program has detected an error condition that prevents it from continuing normal processing.	The conversation's context is placed in the Backout-Required condition.	The program should issue a resource recovery backout call.
	Send_Error[3]	The program has detected an error in received data or some other error that may be correctable.	The SPM backs out the transaction, and both programs are informed of the backout.	Depends on the response from the partner program.
Take-Backout	Commit	This is an error in program logic.	The commit call is treated as though it were a backout call, and the backout request is spread to other programs in the transaction.	None
	Backout	The program agrees to the backout request.	The backout request is spread to other programs in the transaction.	None
	Deallocate (Abend)[2] or Cancel_ Conversation	The program has detected an error condition that prevents it from continuing normal processing.	The conversation's context is placed in the Backout-Required condition.	The program should issue a resource recovery backout call.

[1] If the take-commit indicator ended in *_DATA_OK, the partner may also send data before making any of the other possible responses.
[2] “Deallocate (Abend)” refers to the CPI Communications Deallocate call with a *deallocate_type* of CM_DEALLOCATE_ABEND.
[3] The program can respond with a Send_Error call only when using a half-duplex conversation.

Chained and Unchained Transactions

When a program is using the X/Open TX resource recovery interface, it may choose when the next transaction is started after the current transaction ends. Specifically, if the TX Transaction_Control variable is set to:

- TX_CHAINED, a commit call ends the current transaction and immediately begins the next transaction and establishes a new sync point.
- TX_UNCHAINED, a commit call ends the current transaction but does not begin the next transaction. The program must issue the tx_begin call to the X/Open TX interface to start the next transaction and to establish a new sync point.

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

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

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

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

```
– TX_CHAINED    and  CM_CHAINED_TRANSACTIONS
– TX_CHAINED    and  CM_UNCHAINED_TRANSACTIONS
– TX_UNCHAINED  and  CM_CHAINED_TRANSACTIONS
– TX_UNCHAINED  and  CM_UNCHAINED_TRANSACTIONS
```

Joining a Transaction

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

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

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

   - Allocate
   - Confirm
   - Include_Partner_In_Transaction
   - Prepare
   - Prepare_To_Receive
   - Receive
   - Send_Data
   - Send_Error

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

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

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

Normally, if the remote program receives the request to join the transaction, it joins automatically. When using the X/Open TX resource recovery interface, the program can issue a tx_info() call to see whether or not it is in transaction mode.

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

Notification Received	Possible Response	Reason for Response	Result of Response	Further Action Required
Join- Transaction	tx_begin	The program agrees to join the distributed transaction.	The local program is included in the distributed transaction.	None
	Deallocate (Abend) [1] or Cancel_ Conversation	The program rejects the request to join the distributed transaction.	The local program is included in the distributed transaction.	None

Table 12. Responses to the CM_JOIN_TRANSACTION Indication

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

Superior and Subordinate Programs

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

The **superior** program is the program that initiates the conversation (using the Initialize_Conversation call). A program that issues the Accept_Conversation or Accept_Incoming call is a **subordinate** of the superior program.

Figure 9 shows a commit tree with seven programs participating in the transaction.

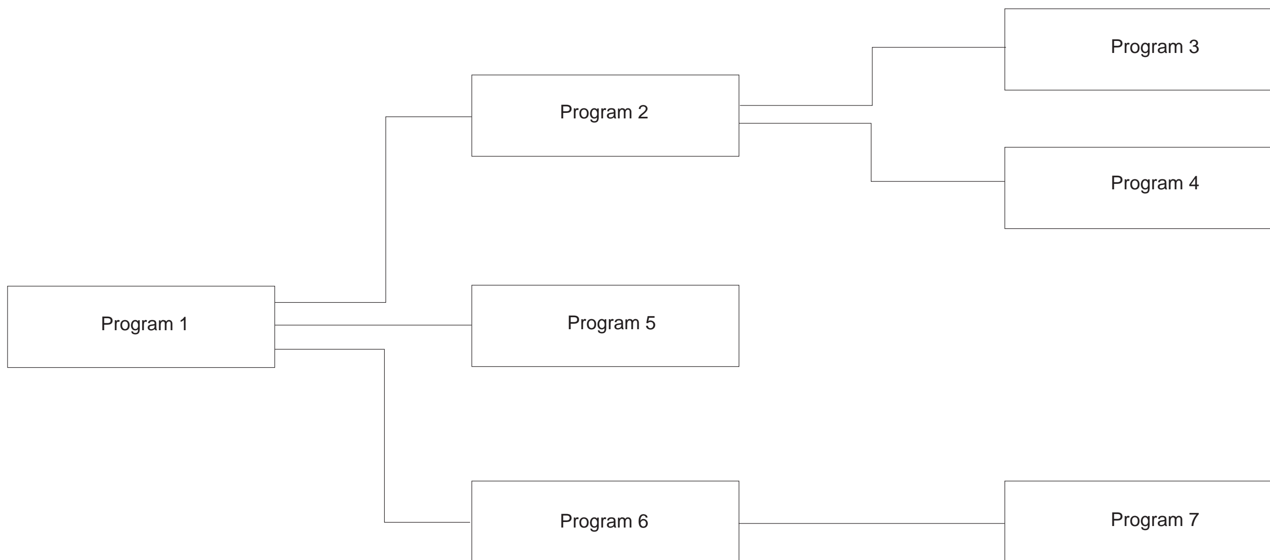


Figure 9. Commit Tree with Program 1 as Root and Superior

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

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

In addition, the Include_Partner_In_Transaction, Prepare, Set_Begin_Transaction, and Set_Prepare_Data_Permitted calls may be issued only by the superior program. These calls return a CM_PROGRAM_PARAMETER_CHECK return code when they are issued by the subordinate.

Additional CPI Communications States

In addition to the conversation states described in "Program Flow—States and Transitions" on page 52, the states described in Table 13 are required when a program uses a protected CPI Communications conversation (that is, with the *sync_level* characteristic set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM).

Table 13 (Page 1 of 2). CPI Communications States for Protected Conversations

State	Description
Defer-Receive	The local program's end of the conversation will enter **Receive** state after a synchronization call completes successfully. The synchronization call may be a resource recovery commit call or a CPI Communications Flush or Confirm call. A conversation enters **Defer-Receive** state when the local program issues a Prepare_To_Receive call with *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, or when it issues a Send_Data call with *send_type* set to CM_SEND_AND_PREP_TO_RECEIVE, *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, and *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM. **Defer-Receive** state is applicable for half-duplex conversations only.
Defer-Deallocate	The local program has requested that the conversation be deallocated after a commit operation has completed; that is, the conversation is included in a transaction, and the program has issued a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, or it has issued a Send_Data call with *send_type* set to CM_SEND_AND_DEALLOCATE, *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL, and *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM. The conversation will not be deallocated until a successful commit operation takes place.
Prepared	The local program has issued a Prepare call to request that the remote program prepare its resources for commitment.

Table 13 (Page 2 of 2). CPI Communications States for Protected Conversations

State	Description
Sync-Point	The local program issued a Receive call and was given a *return_code* of CM_OK and a *status_received* of CM_TAKE_COMMIT or CM_TAKE_COMMIT_DATA_OK. After a successful commit operation, a half-duplex conversation will return to **Receive** state, while a full-duplex conversation will return to **Send-Receive** state.
Sync-Point-Send	The local program issued a Receive call and was given a *return_code* of CM_OK and a *status_received* of CM_TAKE_COMMIT_SEND or CM_TAKE_COMMIT_SEND_DATA_OK. After a successful commit operation, the conversation will be placed in **Send** state. **Sync-Point-Send** state is applicable for half-duplex conversations only.
Sync-Point-Deallocate	The local program issued a Receive call and was given a *return_code* of CM_OK and a *status_received* of CM_TAKE_COMMIT_DEALLOCATE or CM_TAKE_COMMIT_DEALLOC_DATA_OK. After a successful commit operation, the conversation will be deallocated and placed in **Reset** state.

Valid States for Resource Recovery Calls

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

All protected conversations within a context must be in one of the following states for the program to issue a commit call for that context:

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

If a commit call is issued from any other conversation state, the program receives from the resource recovery interface a return code indicating an error. The program can also receive an error return code if the conversation was in **Send** or **Send-Receive** state when the commit call was issued, and the program had started but had not finished sending a basic conversation logical record.

A backout call can be issued in any state.

TX Extensions for CPI Communications

If the subordinate program uses the X/Open TX resource recovery interface and the *join_transaction* characteristic has the value CM_JOIN_IMPLICIT, the TX State Table, in the *X/Open TX Specification*, changes in the following ways:

- An incoming conversation request causes an implicit tx_set_transaction-control() call in the following way:
  - If the value of the CPI-C *transaction_control* characteristic is TX_CHAINED_TRANSACTIONS, the TX *transaction_control* characteristic changes to TX_CHAINED.
  - If the value of the CPI-C *transaction_control* characteristic is TX_UNCHAINED_TRANSACTIONS, the TX *transaction_control* characteristic changes to TX_UNCHAINED.
- An incoming *join_transaction* request causes an implicit tx_begin() call. This causes implicit TX state changes.

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

Chapter 3. Program-to-Program Communication Example Flows

This chapter provides example flows of how two programs using CPI Communications can exchange information and data in a controlled manner.

The examples are divided into these sections:

- “Starter-Set Flows” on page 68
- “Controlling Data Flow Direction” on page 74
- “Verifying Receipt of Data” on page 78
- “Reporting Errors to Partner” on page 80
- “Using Full-Duplex Conversations” on page 84
- “Using Queue-Level Non-Blocking” on page 90
- “Accepting Multiple Conversations” on page 92
- “Using the Distributed Directory” on page 96
- “Resource Recovery Flows” on page 98

In addition to these sample flows, a simple COBOL application using CPI Communications calls is provided on the diskette that came with this manual.

Interpreting the Flow Diagrams

In the flow diagrams shown in this chapter (for example, Figure 10 on page 71), vertical dotted lines indicate the components involved in the exchange of information between systems. The horizontal arrows indicate the direction of the flow for that step. The numbers lined up on the left side of the flow are reference points to the flow and indicate the progression of the calls made on the conversation. These same numbers correspond to the numbers under the **Step** heading of the text description for each example.

The call parameter lists shown in the flows are not complete; only the parameters of particular interest to the flows being discussed are shown. A complete description of each CPI Communications call and the required parameters can be found in Chapter 4, “Call Reference.”

A complete discussion of all possible timing scenarios is beyond the scope of the chapter. Where appropriate, such discussion is provided in the individual call descriptions in Chapter 4, “Call Reference.”

© Copyright IBM Corp. 1996, 1998

Starter-Set Flows

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

- "Example 1: Data Flow in One Direction" on page 69 demonstrates a flow of data in only one direction (only the initiating program sends data).
- "Example 2: Data Flow in Both Directions" on page 72 describes a bidirectional flow of data (the initiating program sends data and then allows the partner program to send data).

Example 1: Data Flow in One Direction

Figure 10 on page 71 shows an example of a conversation where the flow of data is in only one direction.

The steps shown in Figure 10 are:

Step	Description
1	To communicate with its partner program, Program A must first establish a conversation. Program A uses the Initialize_Conversation call to tell CPI Communications that it wants to: • Initialize a conversation • Identify the conversation partner (using *sym_dest_name*) • Ask CPI Communications to establish the identifier that the program will use when referring to the conversation (the *conversation_ID*). Upon successful completion of the Initialize_Conversation call, CPI Communications assigns a *conversation_ID* and returns it to Program A. The program must store the *conversation_ID* and use it on all subsequent calls intended for that conversation.
2	No errors were found on the Initialize_Conversation call, and the *return_code* is set to CM_OK. Two major tasks are now accomplished: • CPI Communications has established a set of conversation characteristics for the conversation, based on the *sym_dest_name*, and uniquely associated them with the *conversation_ID*. • The default values for the conversation characteristics, as listed in “Initialize_Conversation (CMINIT)” on page 200, have been assigned. (For example, the conversation now has *conversation_type* set to CM_MAPPED_CONVERSATION.)
3	Program A asks that a conversation be started with an Allocate call (see “Allocate (CMALLC)” on page 124) using the *conversation_ID* previously assigned by the Initialize_Conversation call.
4	If a logical connection between the systems is not already available, one is activated. Program A and Program C can now have a conversation.
5	A *return_code* of CM_OK indicates that the Allocate call was successful and the system has allocated the necessary resources to the program for its conversation. Program A’s conversation is now in **Send** state and Program A can begin to send data. **Note:** In this example, the error conditions that can arise (such as no logical connections available) are not discussed. See “Allocate (CMALLC)” on page 124 for more information about the error conditions that can result.

Step	Description
6 and 7	Program A sends data with the Send_Data call (described in "Send_Data (CMSEND)" on page 249) and receives a *return_code* of CM_OK. Until now, the conversation may not have been established because the conversation startup request may not be sent until the first flow of data. In fact, any number of Send_Data calls can be issued before CPI Communications actually has a full buffer, which causes it to send the startup request and data. Step 6 shows a case where the amount of data sent by the first Send_Data is greater than the size of the local system's send buffer (a system-dependent property), which is one of the conditions that triggers the sending of data. The request for a conversation is sent at this time. **Notes:** 1. Some implementations may choose to transmit the conversation startup request as part of the Allocate processing. 2. The local program can ensure that the remote program is connected as soon as possible by issuing Flush (CMFLUS) immediately after Allocate (CMALLC). For a complete discussion of transmission conditions and how to ensure the immediate establishment of a conversation and transmission of data, see "Data Buffering and Transmission" on page 44.
8 and 9	Once the conversation is established, the remote program's system takes care of starting Program C. The conversation on Program C's side is in **Reset** state and Program C issues a call to Accept_Conversation, which places the conversation into **Receive** state. The Accept_Conversation call is similar to the Initialize_Conversation call in that it equates a *conversation_ID* with a set of conversation characteristics (see "Accept_Conversation (CMACCP)" on page 119 for details). Program C, like Program A in Step 2, receives a unique *conversation_ID* that it will use in all future CPI Communications calls for that particular conversation. As discussed in "Conversation Characteristics" on page 33, some of Program C's defaults are based on information contained in the conversation startup request.
10 and 11	Once its end of the conversation is in **Receive** state, Program C begins whatever processing role it and Program A have agreed upon. In this case, Program C accepts data with a Receive call (as described in "Receive (CMRCV)" on page 213). Program A could continue to make Send_Data calls (and Program C could continue to make Receive calls), but, for the purposes of this example, assume that Program A only wanted to send the data contained in its initial Send_Data call.
12	Program A issues a Deallocate call (see "Deallocate (CMDEAL)" on page 143) to send any data buffered in the local system and release the conversation. Program C issues a final Receive, shown here in the same step as the Deallocate, to check that it has all the received data.
13 and 14	The *return_code* of CM_DEALLOCATED_NORMAL tells Program C that the conversation is deallocated. Both Program C and Program A finish normally. **Note:** Only one program should issue Deallocate; in this case it was Program A. If Program C had issued Deallocate after receiving CM_DEALLOCATED_NORMAL, an error would have resulted.

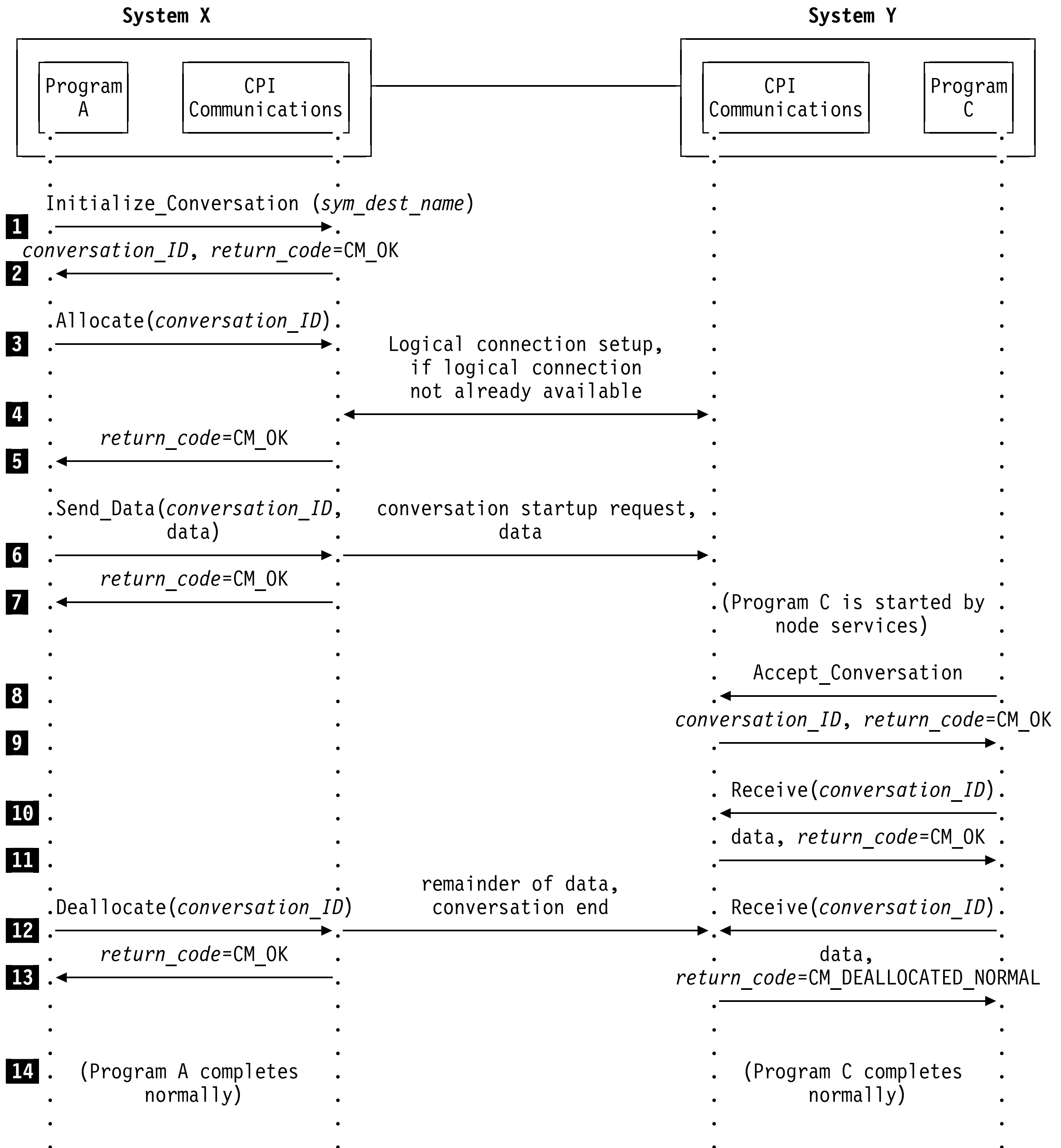


Figure 10. Data Flow in One Direction

Example 2: Data Flow in Both Directions

Figure 11 shows a conversation in which the flow of data is in both directions. It describes how two programs using starter-set calls can initiate a change of control over who is sending the data.

The steps shown in Figure 11 are:

Step	Description
1 through **4**	Program A is sending data and Program C is receiving data. **Note:** The conversation in this example is already established with the default characteristics. Program A's end of the conversation is in **Send** state, and Program C's is in **Receive** state.
5	After sending some amount of data (an indeterminate number of Send_Data calls), Program A issues the Receive call while its end of the conversation is in **Send** state. As described in "Receive (CMRCV)" on page 213, this call causes the remaining data buffered at System X to be sent and permission to send to be given to Program C. Program A's end of the conversation is placed in **Receive** state, and Program A waits for a response from Program C. **Note:** See "Example 3: The Sending Program Changes the Data Flow Direction" on page 74 for alternate methods that allow Program A to continue processing. Program C issues a Receive call in the same way it issued the two prior Receive calls.
6	Program C receives not only the last of the data from Program A, but also a *status_received* parameter set to CM_SEND_RECEIVED. The value of CM_SEND_RECEIVED notifies Program C that its end of the conversation is now in **Send** state.
7	As a result of the *status_received* value, Program C issues a Send_Data call. The data from this call, on arrival at System X, is returned to Program A as a response to the Receive it issued in Step **5**. At this point, the flow of data has been completely reversed and the two programs can continue whatever processing their logic dictates. To give control of the conversation back to Program A, Program C would simply follow the same procedure that Program A executed in Step **5**.
8 through **10**	Programs A and C continue processing. Program C sends data and Program A receives the data.

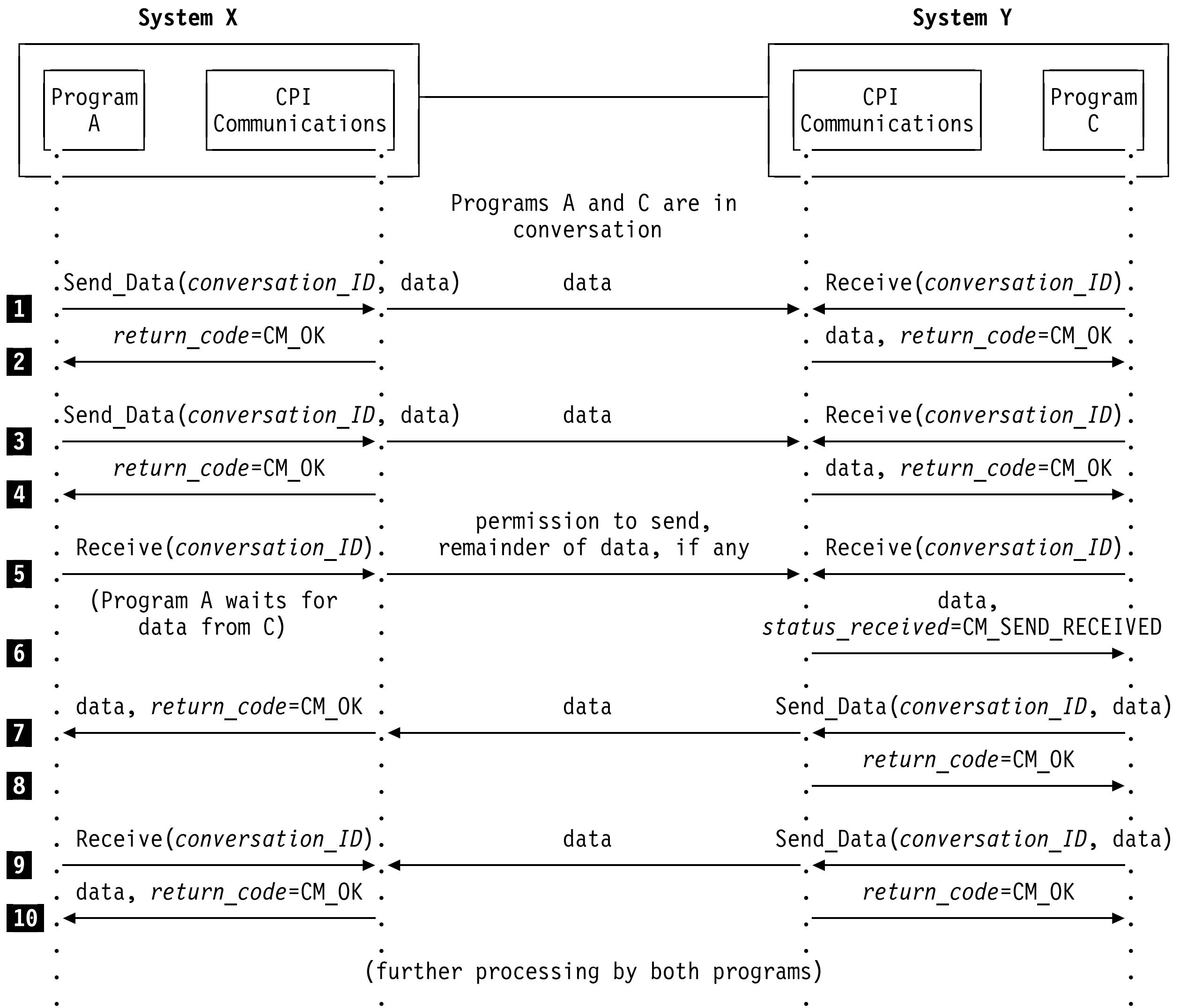


Figure 11. Data Flow in Both Directions

Controlling Data Flow Direction

This section discusses how a program can exercise control over the data flow direction:

- "Example 3: The Sending Program Changes the Data Flow Direction" shows how to use the Prepare_To_Receive call to change the direction of the data flow on a half-duplex conversation.
- "Example 4: The Receiving Program Changes the Data Flow Direction" shows how to use the Request_To_Send call to request a change in the direction of the data flow on a half-duplex conversation.

Example 3: The Sending Program Changes the Data Flow Direction

Figure 12 is a variation on the function provided by the flow shown in "Example 2: Data Flow in Both Directions" on page 72. When the data flow direction changes, Program A can continue processing instead of waiting for data to arrive on this half-duplex conversation.

The steps shown in Figure 12 are:

Step	Description
1 through **6**	The program begins the same as "Example 1: Data Flow in One Direction" on page 69. Program A establishes the conversation and makes the initial transmission of data.
7 through **10**	Program A makes use of an advanced-function call, Prepare_To_Receive, (described in "Prepare_To_Receive (CMPTR)" on page 208), which sends an indication to Program C that Program A is ready to receive data. This call also flushes the data buffer and places Program A's end of the conversation into **Receive** state. It does not, as did the Receive call when used with the initial conversation characteristics in effect, force Program A to pause and wait for data from Program C to arrive. Program A continues processing while data is sent to Program C.
11 through **13**	Program C, started by System Y's reception of the conversation startup request and buffered data, makes the Accept_Conversation and Receive calls. Program A finishes its processing and issues its own Receive call. It will now wait until data is received (Step **15**).
14 through **16**	The *status_received* on the Receive call made by Program C, which is set to CM_SEND_RECEIVED, tells Program C that the conversation is in **Send** state. Program C can now issue the Send_Data call. Program A receives the data. **Note:** There is a way for Program A to check periodically to see if the data has arrived, without waiting. After issuing the Prepare_To_Receive call, Program A can use the Set_Receive_Type call to set the *receive_type* conversation characteristic equal to CM_RECEIVE_IMMEDIATE. This call changes the nature of all subsequent Receive calls issued by Program A (until a further call to Set_Receive_Type is made). If a Receive is issued with the *receive_type* set to CM_RECEIVE_IMMEDIATE, the program retains control of processing without waiting. It receives data back if data is present, and a *return_code* of CM_UNSUCCESSFUL if no data has arrived. This method of receiving data is not shown in Figure 12. For further discussion of this alternate flow, see "Set_Receive_Type (CMSRT)" on page 344 and "Receive (CMRCV)" on page 213.

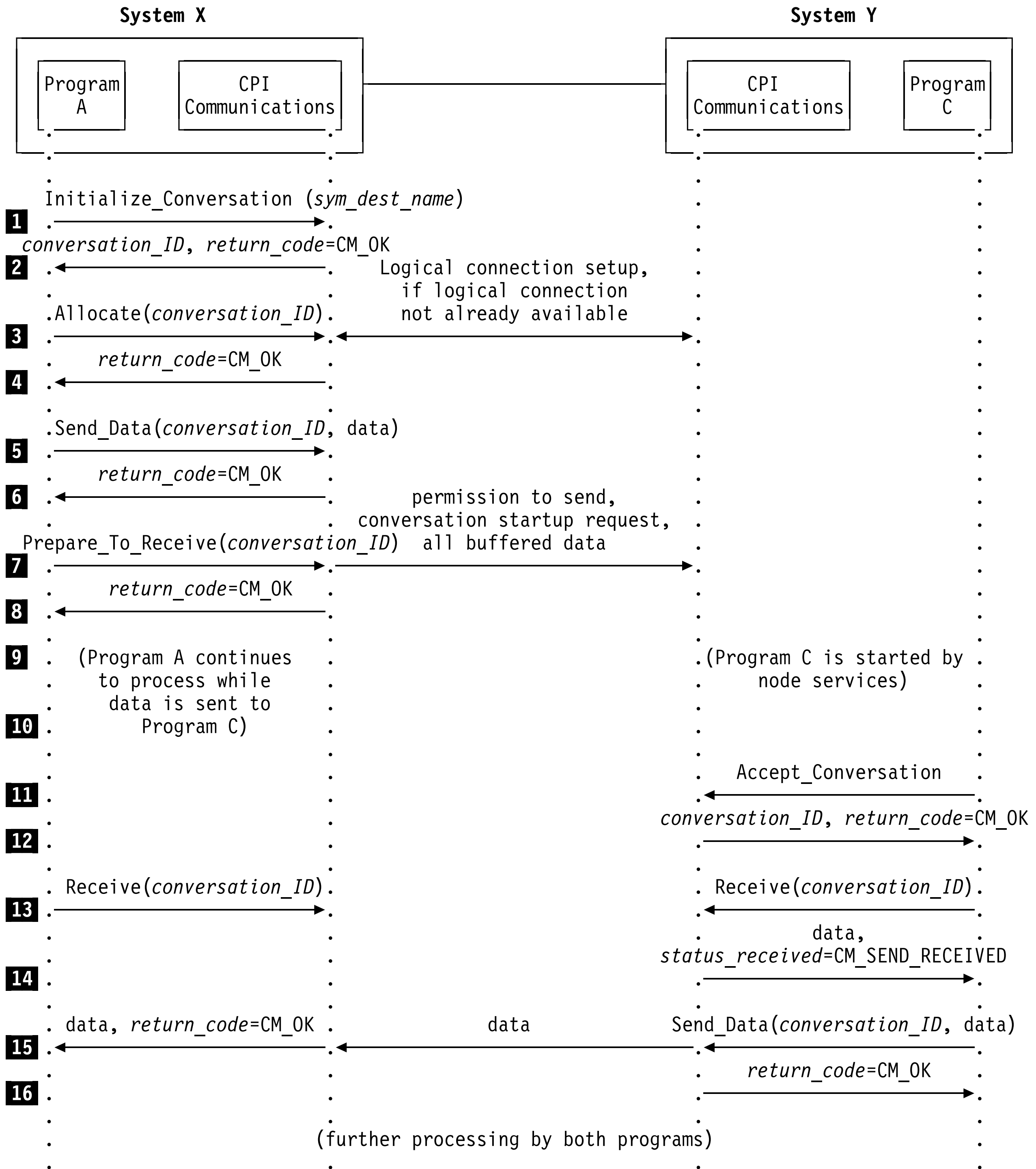


Figure 12. The Sending Program Changes the Data Flow Direction

Example 4: The Receiving Program Changes the Data Flow Direction

Figure 13 shows how a program on the receiving side of a half-duplex conversation can request a change in the direction of data flow with the Request_To_Send call. (See "Request_To_Send (CMRTS)" on page 246 for more information.) In this example, Programs A and C have already established a conversation using the default conversation characteristics.

The steps shown in Figure 13 are:

Step	Description
1 and **2**	Program A is sending data and Program C is receiving the data.
3 and **4**	Program C issues a Request_To_Send call in order to begin sending data. Program A will be notified of this request on the return value of the next call issued by Program A (Send_Data in this case, Step **6**).
5 and **6**	Program A issues a Send_Data request, and the call returns with *control_information_received* set equal to CM_REQ_TO_SEND_RECEIVED.
7 and **8**	In reply to the Request_To_Send, Program A issues a Prepare_To_Receive call, which allows Program A to continue its own processing and passes permission to send to Program C. The call also forces the buffer at System X to be flushed. It leaves the conversation in **Receive** state for Program A.
	Note: Program A does not have to reply to the Request_To_Send call immediately (as it does in this example). See "Example 3: The Sending Program Changes the Data Flow Direction" on page 74 for other possible responses.
	Program C continues with normal processing by issuing a Receive call and receives Program A's acceptance of the Request_To_Send on the *status_received* parameter, which is set to CM_SEND_RECEIVED. The conversation is now in **Send** state for Program C.
9 and **10**	Program C can now transmit data. Because Program C has only one instance of data to transmit, it first changes the *send_type* conversation characteristic by issuing Set_Send_Type. Setting *send_type* to a value of CM_SEND_AND_PREP_TO_RECEIVE means that Program C's end of the conversation will return to **Receive** state after Program C issues a Send_Data call. It also forces a flushing of the system's data buffer.
11	Program C issues the Send_Data call and its end of the conversation is placed in **Receive** state. The data and permission-to-send indication are transmitted from System Y to System X.
	Program A, meanwhile, has finished its own processing and issued a Receive call (which is perfectly timed, in this diagram).
12	Program A receives the data requested and, because of the value of the *status_received* parameter (which is set to CM_SEND_RECEIVED), knows that the conversation has been returned to **Send** state.
13 and **14**	The original processing flow continues: Program A issues a Send_Data call and Program C issues a Receive call.

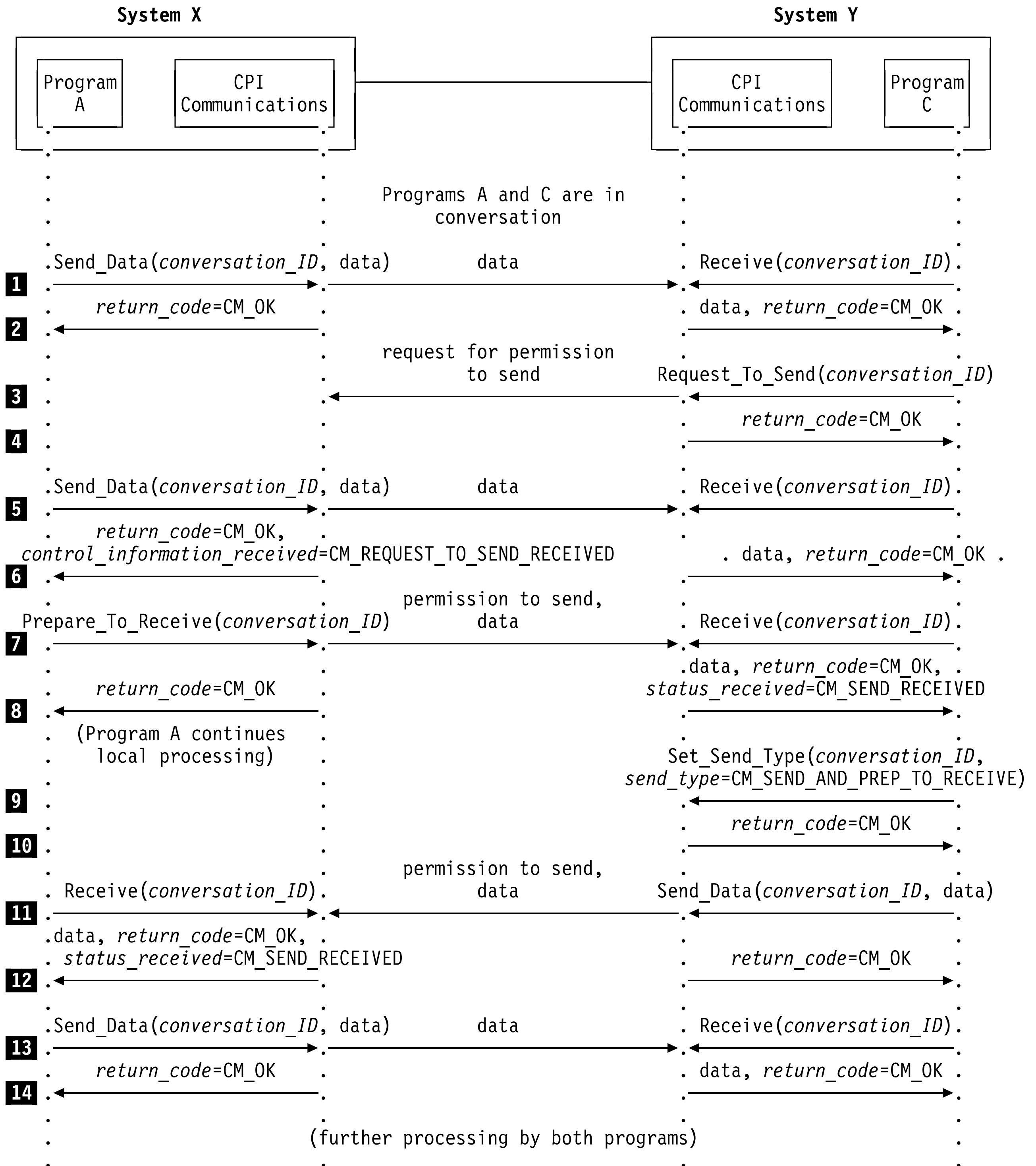


Figure 13. Changing the Data Flow Direction

Verifying Receipt of Data

This section provides an example of a program validating receipt of data.

Example 5: Validation of Data Receipt

Figure 14 on page 79 shows how a program can use the Confirm and Confirmed calls on a half-duplex conversation to verify receipt of its sent data. The Flush call is also shown.

The steps shown in Figure 14 are:

Step	Description
1 and 2	As before, Program A issues the Initialize_Conversation call to initialize the conversation.
3 and 4	Program A issues a new call, Set_Sync_Level, to set the *sync_level* characteristic to CM_CONFIRM. **Note:** Program A must set the *sync_level* characteristic before issuing the Allocate call (Step 5) for the value to take effect. Attempting to change the *sync_level* after the conversation is allocated results in an error condition. See "Set_Sync_Level (CMSSL)" on page 354 for a detailed discussion of the *sync_level* characteristic and the meaning of CM_CONFIRM.
5 and 6	Program A issues the Allocate call to start the conversation.
7 and 8	Program A uses the Flush call (see "Flush (CMFLUS)" on page 195) to make sure that the conversation is immediately established. If data is present, the local system buffer is emptied and the contents are sent to the remote system. Since no data is present, only the conversation startup request is sent to establish the conversation. At System Y, the conversation startup request is received. Program C is started and begins processing.
9 and 10	Program A issues a Send_Data call. Program C issues an Accept_Conversation call.
11	Program A issues a Confirm call to make sure that Program C has received the data and performed any data validation that Programs A and C have agreed upon. Program A is forced to wait for a reply.
12 and 13	Program C issues a Receive call and receives the data with *status_received* set to CM_CONFIRM_RECEIVED.
14 and 15	Because *status_received* is set to CM_CONFIRM_RECEIVED, indicating a confirmation request, the conversation has been placed into **Confirm** state. Program C must now issue a Confirmed call. After Program C makes the Confirmed call (see "Confirmed (CMCFMD)" on page 137), the conversation returns to **Receive** state. Meanwhile, at System X, the confirmation reply arrives and the CM_OK *return_code* is sent back to Program A.
16	Program A continues with further processing. **Note:** Unlike the previous examples in which a program could bypass waiting, this example demonstrates that use of the Confirm call forces the program to wait for a reply.

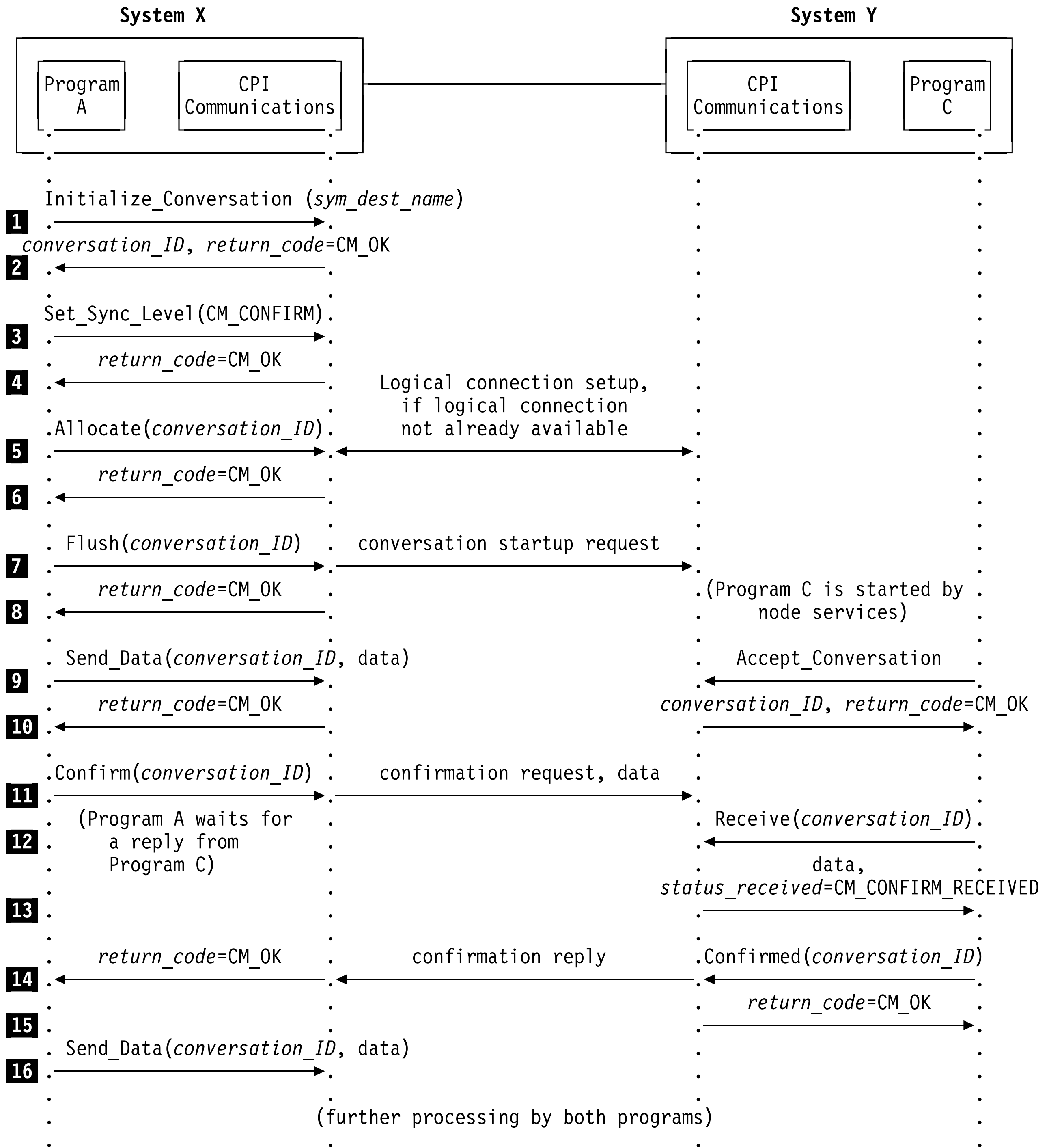


Figure 14. Validation and Confirmation of Data Reception

Reporting Errors to Partner

All the previous examples assumed that no errors were found in the data, and that the receiving program was able to continue receiving data. However, in some cases the local program may detect an error in the data or may find that it is unable to receive more data (for example, its buffers are full) and cannot wait for the remote program to honor a request-to-send request.

This section provides two examples of error reporting:

- "Example 6: Reporting Errors" shows how to use the Send_Error call to report errors in the data flow on a half-duplex conversation.
- "Example 7: Error Direction and Send-Pending State" shows how to use the **Send-Pending** state and the *error_direction* characteristic to resolve an ambiguous error condition that can occur when a program receives both a change-of-direction indication and data for a Receive call on a half-duplex conversation.

Example 6: Reporting Errors

Figure 15 is an example of programs using a half-duplex conversation. This example describes the simplest type of error reporting, an error found while receiving data. "Example 7: Error Direction and Send-Pending State" on page 82 describes a more complicated use of Send_Error.

The steps shown in Figure 15 are:

Step	Description
1 and 2	Program A is sending data and Program C is receiving data. The initial characteristic values set by Initialize_Conversation and Accept_Conversation are in effect.
3 and 4	Program C encounters an error on the received data and issues the Send_Error call. The local system sends control information to System X indicating that the Send_Error has been issued and purges all data contained in its buffer.
5 and 6	Meanwhile, Program A has sent more data. This data is purged because System X knows that a Send_Error has been issued at System Y (the control information sent in Step 3). After System X sends control information to System Y, a *return_code* of CM_OK is returned to Program C and the conversation is left in **Send** state. Program A learns of the error (and possibly lost data) when it receives back the *return_code*, which is set to CM_PROGRAM_ERROR_PURGING. Program A's end of the conversation is also placed into **Receive** state, in a parallel action to the now-new **Send** state of the conversation for Program C.
7 and 8	Program C issues a Send_Data call, and Program A receives the data using the Receive call. Programs A and C continue processing normally.

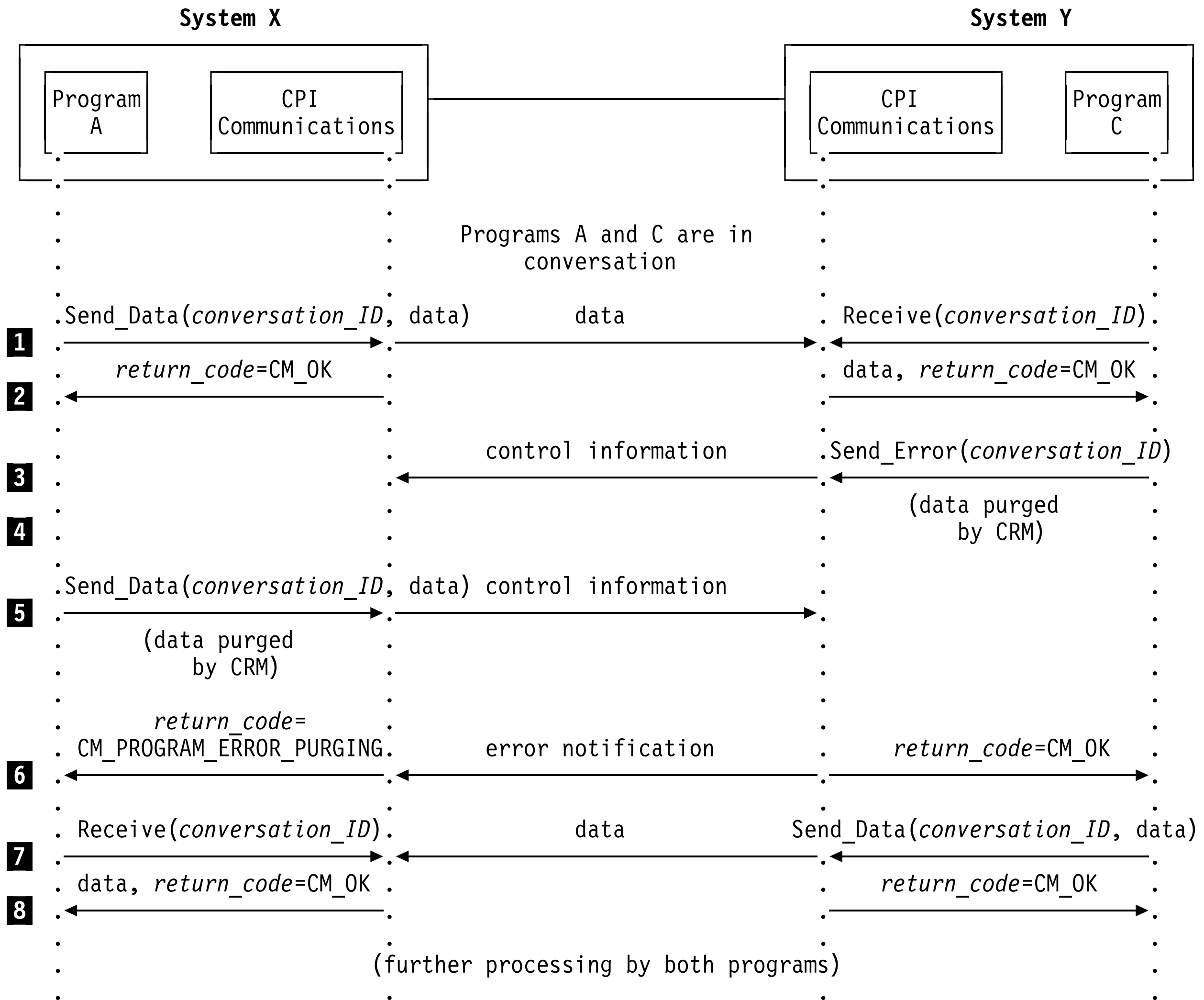


Figure 15. Reporting Errors

Example 7: Error Direction and Send-Pending State

Figure 16 on page 83 shows how to use the **Send-Pending** state and the *error_direction* characteristic to resolve an ambiguous error condition that can occur when a program receives both a change of direction indication and data on a Receive call.

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

The steps shown in Figure 16 are:

Step	Description
1 and 2	The conversation has already been established using the default conversation characteristics. Program A is sending data in **Send** state and Program C is receiving data in **Receive** state.
3	Program A issues the Receive call to begin receiving data and its end of the conversation enters **Receive** state.
4 and 5	Program C issues a Receive and is notified of the change in the conversation's state by the *status_received* parameter, which is set to CM_SEND_RECEIVED. The reception of both data and CM_SEND_RECEIVED on the same Receive call places Program C's end of the conversation into **Send-Pending** state. Two possible error conditions can now occur: • Program C, while processing the data just received, discovers something wrong with the data (as was discussed in "Example 6: Reporting Errors"). This is an error in the "receive" direction of the data. • Program C finishes processing the data and begins its send processing. However, it discovers that it cannot send a reply. For example, the received data might contain a query for a particular database. Program C successfully processes the query but finds that the database is not available when it attempts to access that database. This is an error in the "send" direction of the data. The *error_direction* characteristic is used to indicate which of these two conditions has occurred. A program sets *error_direction* to CM_RECEIVE_ERROR for the first case and sets *error_direction* to CM_SEND_ERROR for the second.
6 and 7	In this example, Program C encounters a send error and issues Set_Error_Direction to set the *error_direction* characteristic to CM_SEND_ERROR. **Note:** The *error_direction* characteristic was not set in the previous example because Program C did not receive send control with the data and, consequently, the conversation did not enter **Send-Pending** state. The *error_direction* characteristic is relevant only when the conversation is in **Send-Pending** state.
8	Program C issues Send_Error. Because CPI Communications knows the conversation is in **Send-Pending** state, it checks the *error_direction* characteristic and notifies the CPI Communications component at System X which type of error has occurred. Program A receives the error information in the *return_code*. The *return_code* is set to CM_PROGRAM_NO_TRUNC because Program C set *error_direction* to CM_SEND_ERROR. If *error_direction* had been set to CM_RECEIVE_ERROR, Program A would have received a *return_code* of CM_PROGRAM_ERROR_PURGING (as in the previous example).
9 through 11	Program C notifies Program A of the exact nature of the problem and both programs continue processing.

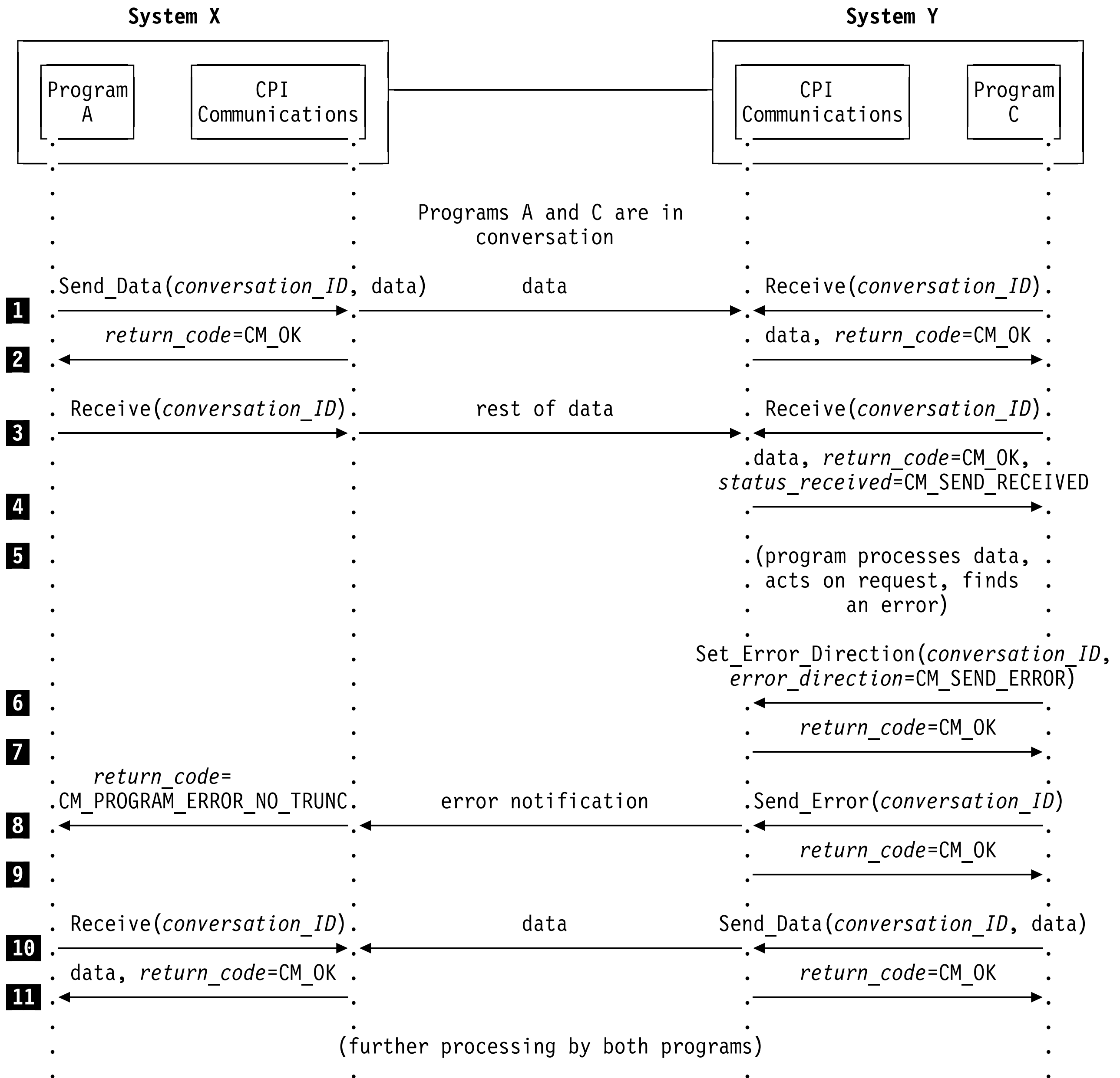


Figure 16. Error Direction and Send-Pending State

Using Full-Duplex Conversations

This section provides examples of programs using full-duplex conversation flows:

- "Example 8: Establishing a Full-Duplex Conversation" describes how a full-duplex conversation is established.
- "Example 9: Using a Full-Duplex Conversation" on page 86 describes how a full-duplex conversation is used to send and receive data.
- "Example 10: Terminating a Full-Duplex Conversation" on page 88 describes how a full-duplex conversation can be terminated.

Example 8: Establishing a Full-Duplex Conversation

Figure 17 on page 85 is an example of how a full-duplex conversation is set up.

The steps shown in Figure 17 are:

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

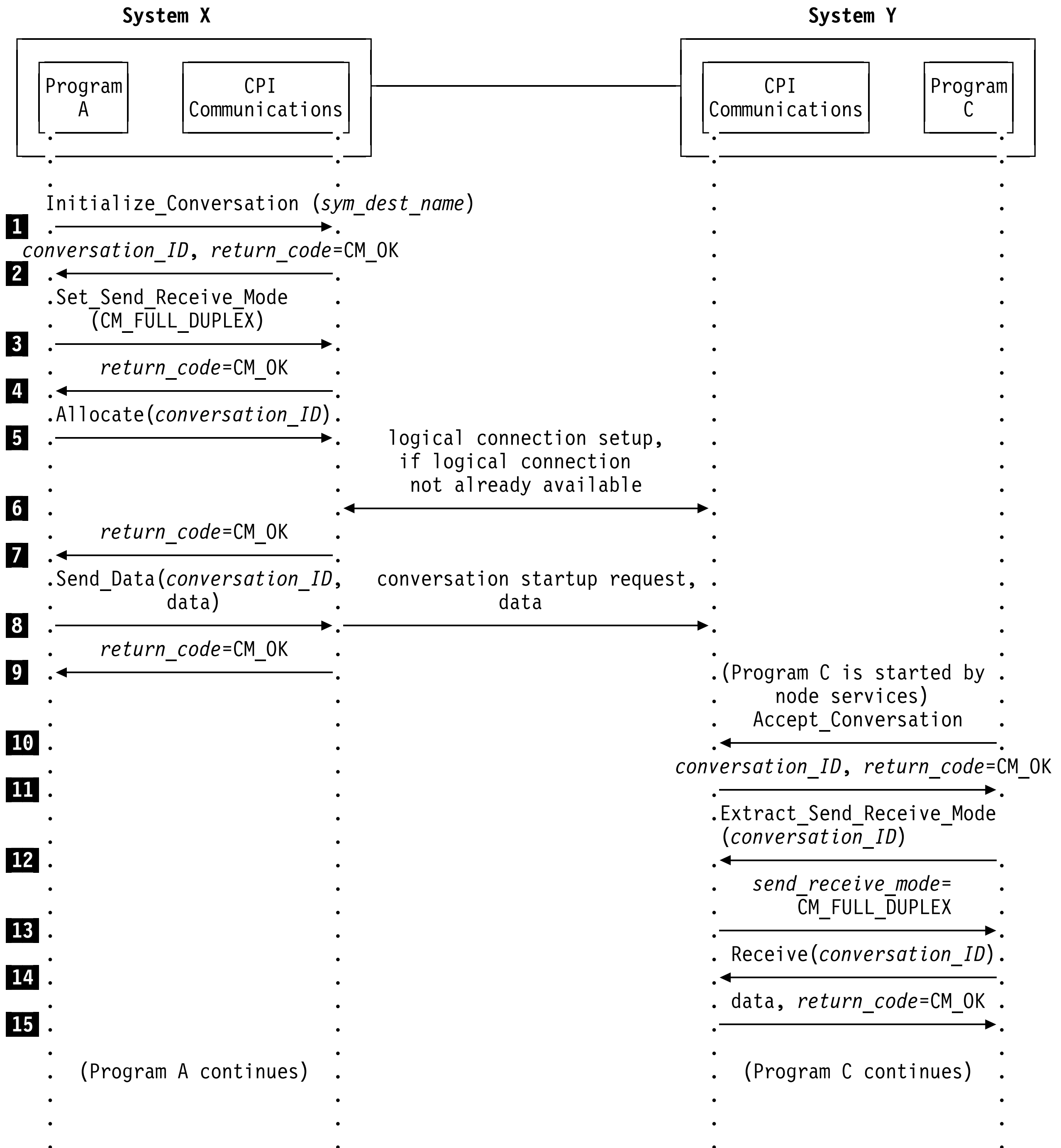


Figure 17. Establishing a Full-Duplex Conversation

Example 9: Using a Full-Duplex Conversation

Figure 18 on page 87 shows an example of how a full-duplex conversation is used to send and receive data.

The steps shown in Figure 18 are:

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

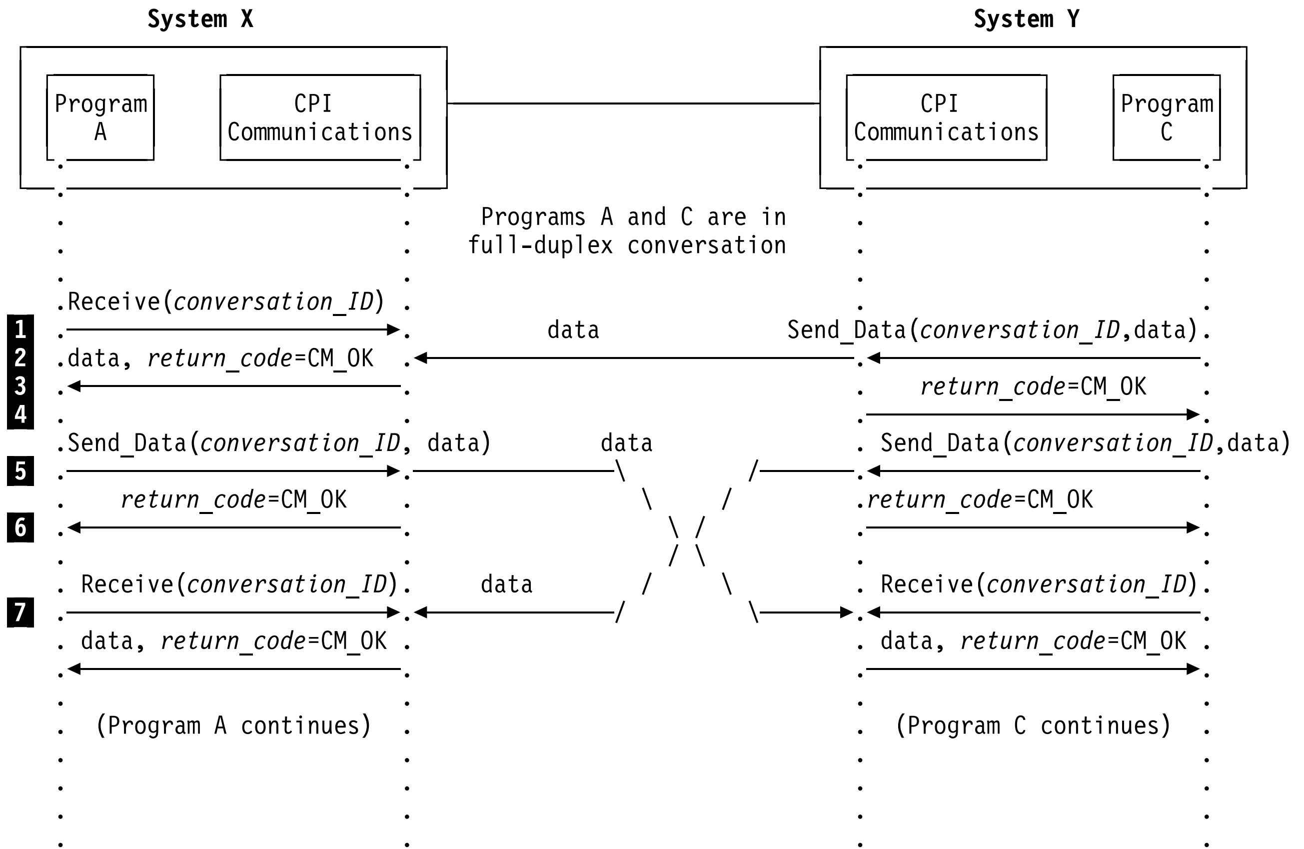


Figure 18. Using a Full-Duplex Conversation

Example 10: Terminating a Full-Duplex Conversation

Figure 19 on page 89 shows an example of how a full-duplex conversation can be terminated.

The steps shown in Figure 19 are:

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

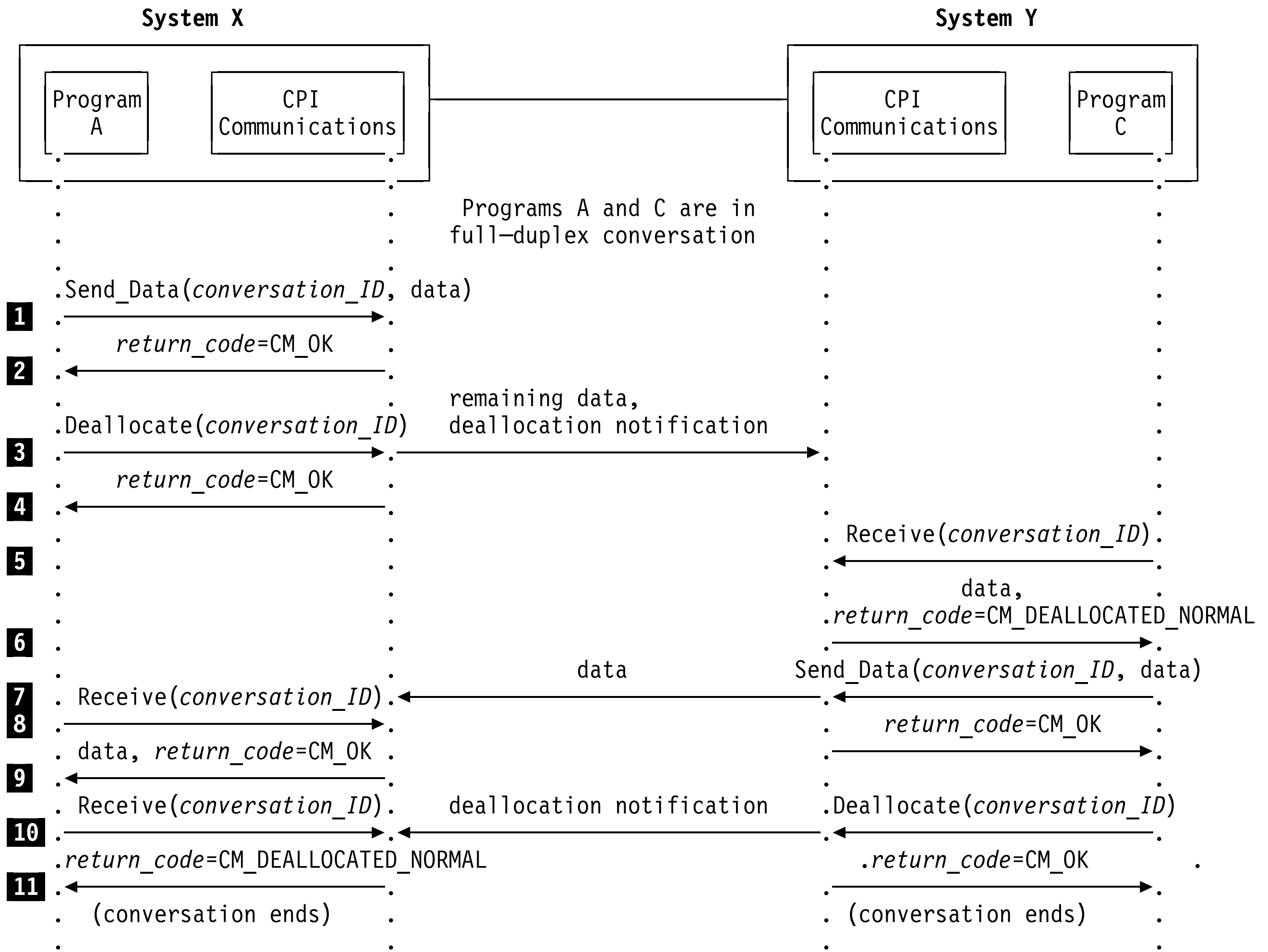


Figure 19. Terminating a Full-Duplex Conversation

Using Queue-Level Non-Blocking

This section provides an example of a program using queue-level non-blocking.

Example 11: Queue-Level Non-Blocking

Figure 20 on page 91 shows an example of a program that uses queue-level non-blocking.

The steps shown in Figure 20 are:

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

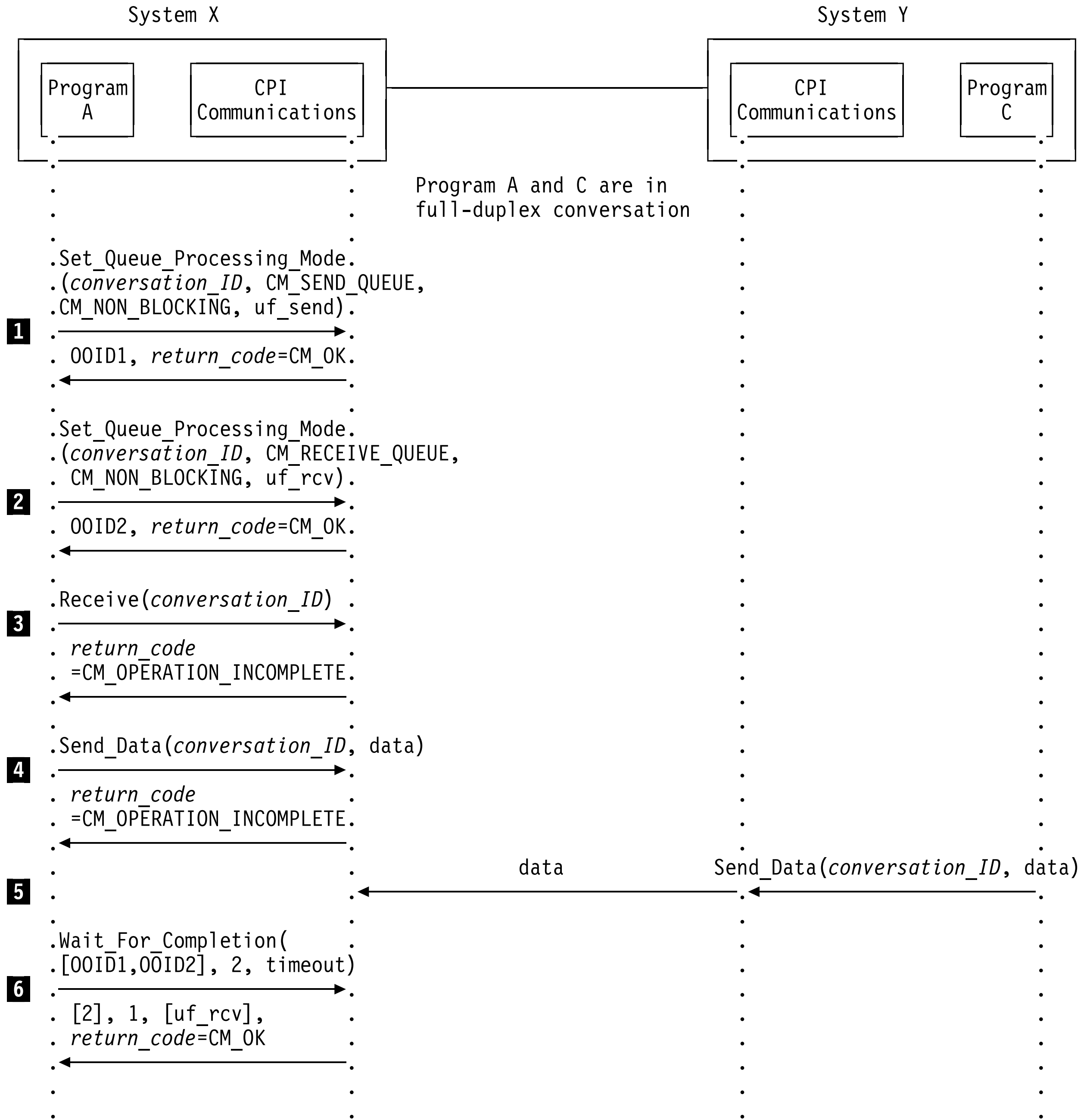


Figure 20. Using Queue-Level Non-Blocking

Accepting Multiple Conversations

This section provides examples of programs accepting multiple conversations:

- "Example 12: Accepting Multiple Conversations Using Blocking Calls" shows a program that uses blocking calls to accept multiple incoming half-duplex conversations.
- "Example 13: Accepting Multiple Conversations Using Conversation-Level Non-Blocking Calls" shows a program that uses non-blocking calls to accept multiple incoming half-duplex conversations.

Example 12: Accepting Multiple Conversations Using Blocking Calls

Figure 21 on page 93 shows an example of a program that uses blocking calls to accept multiple incoming half-duplex conversations.

The steps shown in Figure 21 are:

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

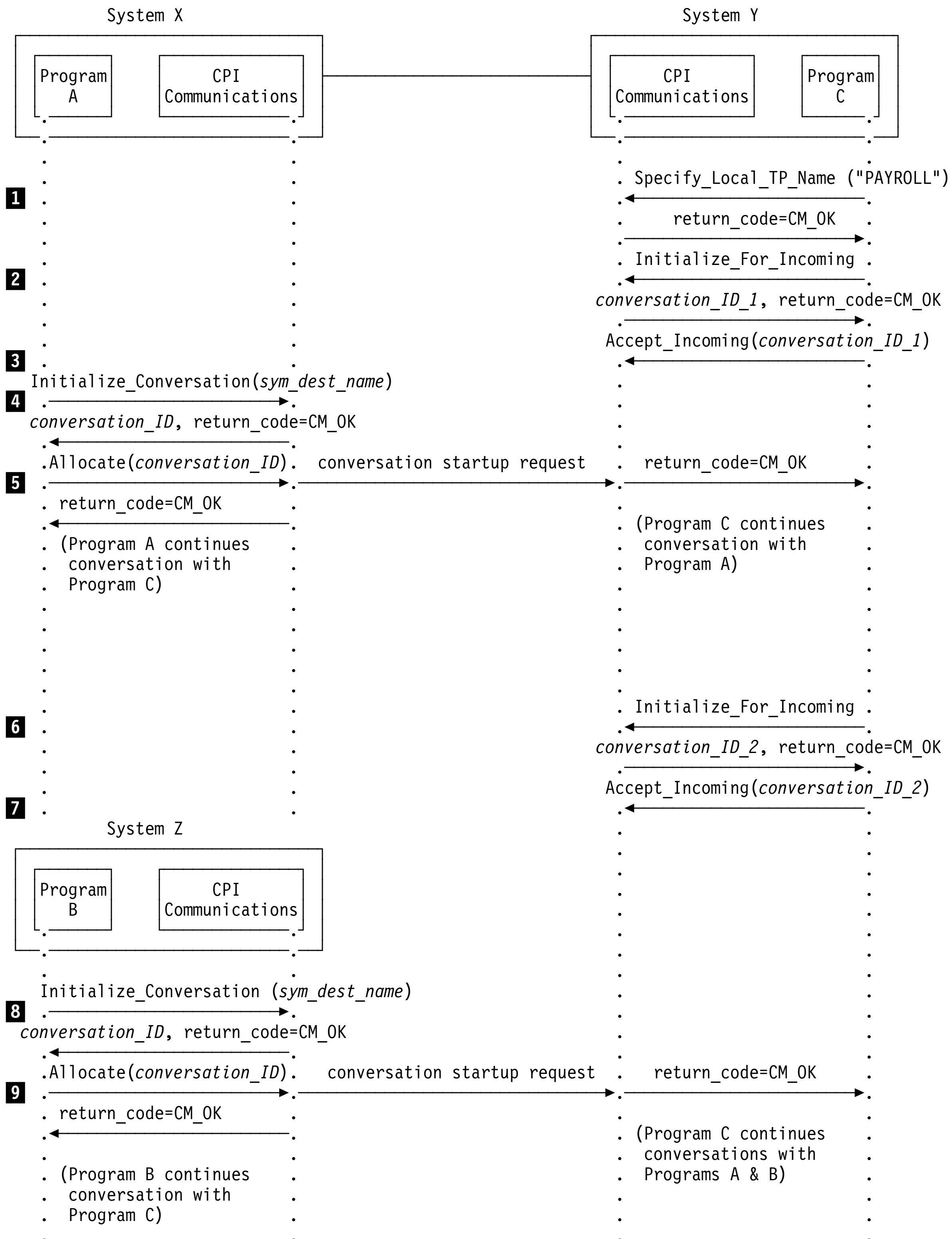


Figure 21. Accepting Multiple Conversations Using Blocking Calls

Example 13: Accepting Multiple Conversations Using Conversation-Level Non-Blocking Calls

Figure 22 on page 95 shows an example of a program that uses conversation-level non-blocking calls to accept multiple incoming half-duplex conversations.

The steps shown in Figure 22 are:

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

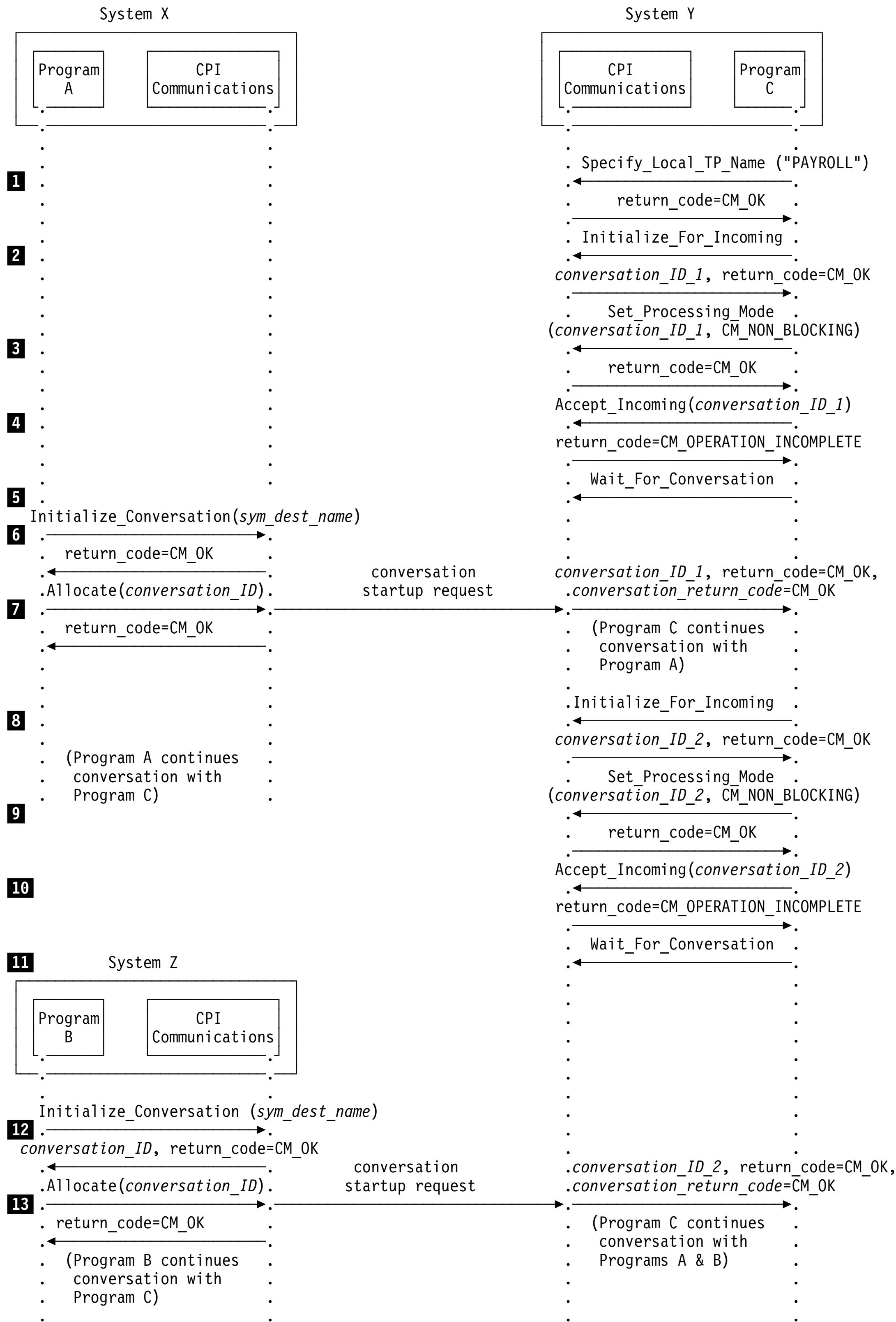


Figure 22. Accepting Multiple Conversations Using Non-Blocking Calls

Using the Distributed Directory

This section provides an example of a program using the distributed directory to locate the partner program.

Example 14: Using the Distributed Directory to Find the Partner Program

Figure 23 is a variation on the function provided by the flow shown in "Example 1: Data Flow in One Direction" on page 69. In this case, the program directly provides a distinguished name so as to access information contained in the distributed directory.

The steps shown in Figure 23 are:

Step	Description
1 and **2**	Program A establishes the conversation and retrieves a *conversation_ID* using a *sym_dest_name* of all blanks.
3 and **4**	Program A uses the Set_Partner_ID call to provide a distinguished name to CPI Communications. To provide a program binding or PFID, Program A could use the same call and set the *partner_ID_type* parameter to CM_PROGRAM_BINDING or CM_PROGRAM_FUNCTION_ID.
5 and **6**	When Program A issues the Allocate call, CPI Communications uses the distinguished name it received in **3** to retrieve a program binding from the distributed directory. This information is then used to establish a logical connection with the partner CRM and allocate a conversation with Program C.
7 and **8**	After the conversation has been successfully allocated, the program can issue the Extract_Partner_ID call to retrieve the specific program binding, as shown in Step **7**. The *partner_ID* parameter returned in Step **8** contains the program binding used to allocate the conversation with Program C. Note that the *partner_ID* field, which contained a distinguished name prior to the Allocate, has now been updated to contain a program binding.
9 and **10**	Program A begins to send data to its partner program and continues with further processing.

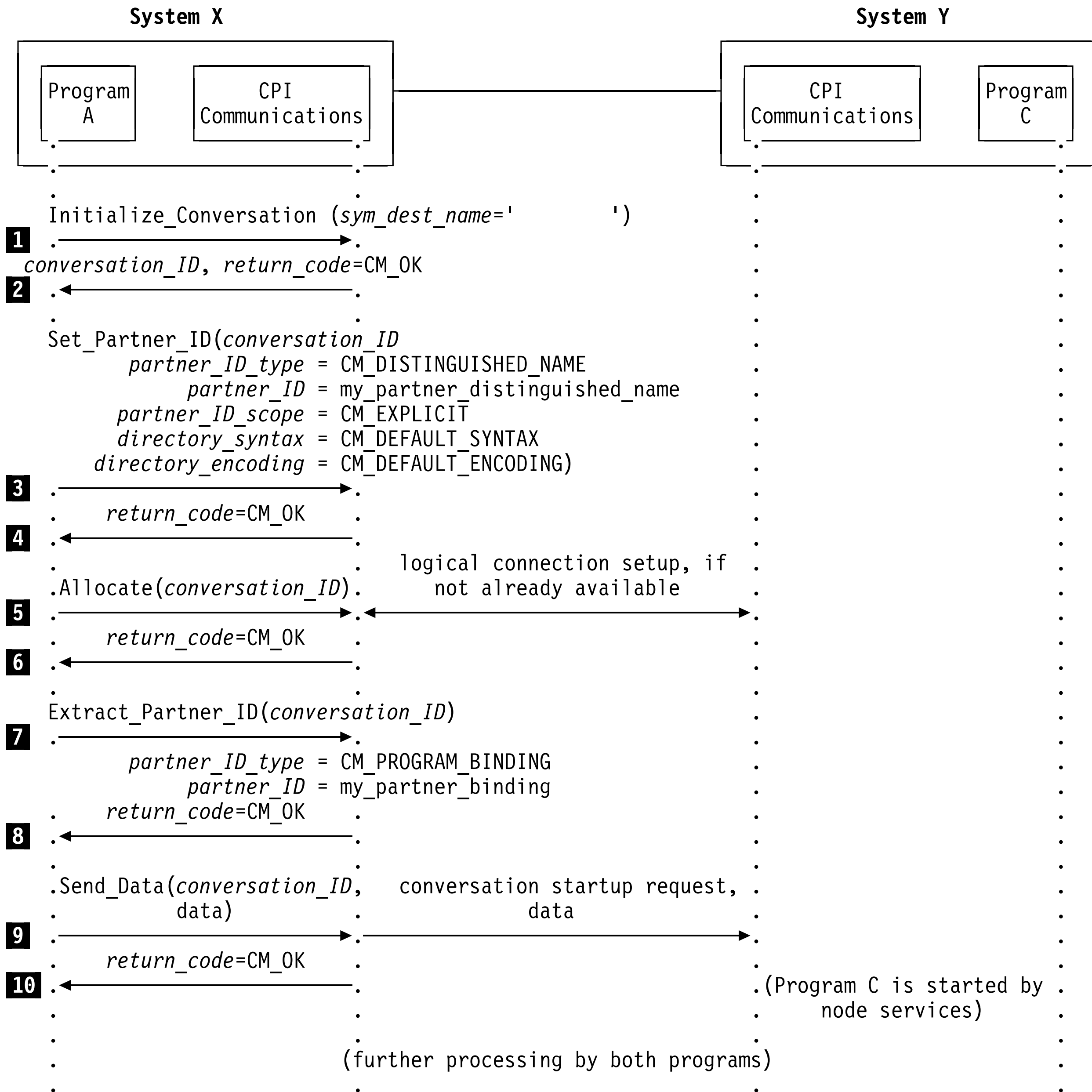


Figure 23. Using the Distributed Directory to Locate the Partner Program

Resource Recovery Flows

This section provides examples of resource recovery flows. These examples use CPI Communications with the resource recovery commit call. The acronym **RR/SPM** is used in the following examples to represent the resource recovery interface functioning in conjunction with a sync point manager. The actual resource recovery call names and return codes depend on the resource recovery interface being used.

- "Example 15: Sending Program Issues a Commit" shows how to use a protected conversation with a resource recovery commit call to establish a synchronization point for protected resources on a half-duplex conversation.
- "Example 16: Successful Commit with Conversation State Change" shows a successful resource recovery commit operation with a conversation state change on a half-duplex conversation.
- "Example 17: Conversation Deallocation before the Commit Call" shows a resource recovery commit call issued after a CPI Communications Deallocate call on a half-duplex conversation.

Example 15: Sending Program Issues a Commit

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

The steps shown in Figure 24 are:

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

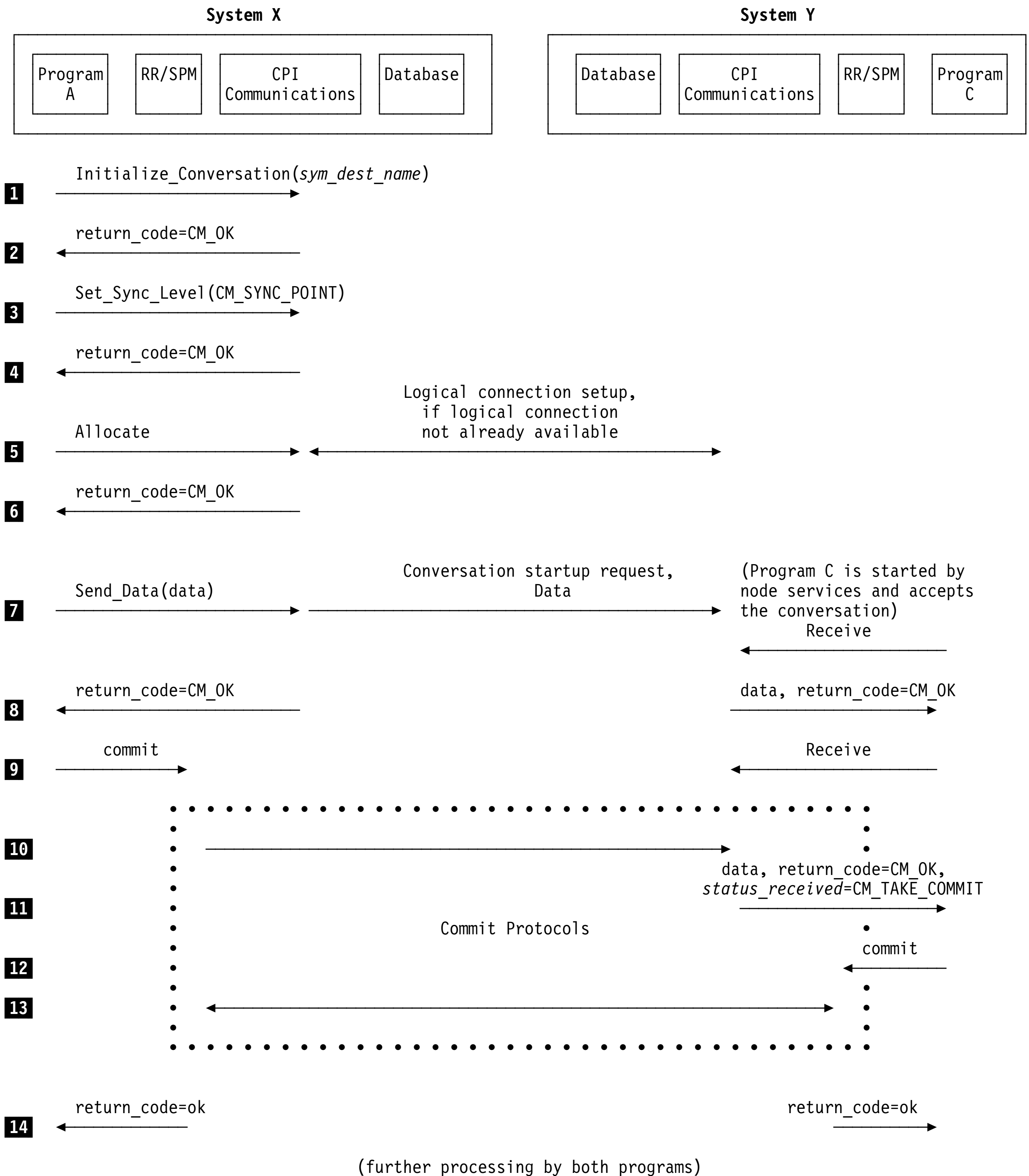


Figure 24. Establishing a Protected Conversation and Issuing a Successful Commit

Example 16: Successful Commit with Conversation State Change

Figure 25 on page 101 shows a successful commit with a conversation state change on a half-duplex conversation.

The steps shown in Figure 25 are:

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

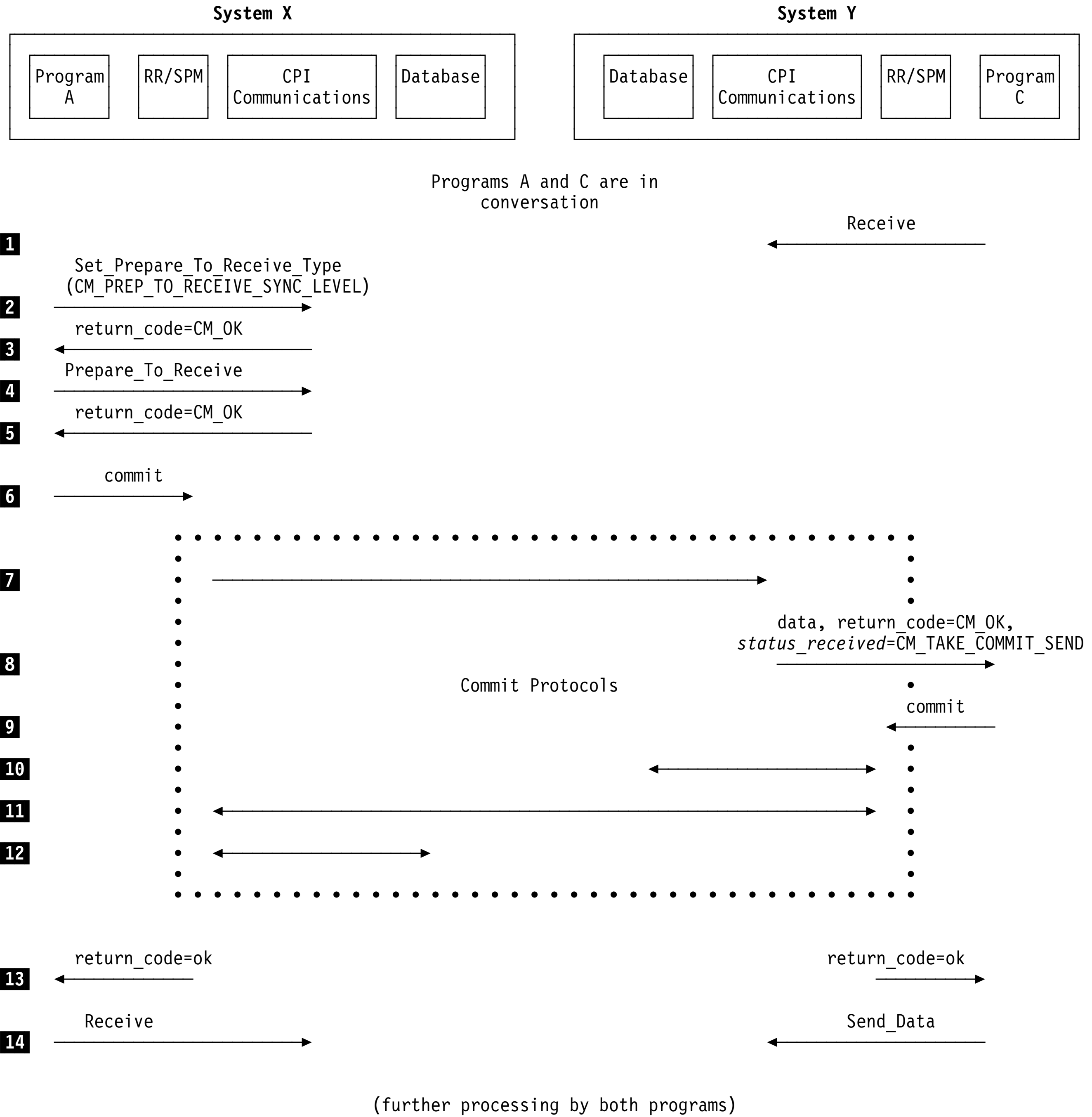


Figure 25. A Successful Commit with Conversation State Change

Example 17: Conversation Deallocation before the Commit Call

Figure 26 on page 103 shows a commit call issued after a CPI Communications Deallocate call on a half-duplex conversation.

The steps shown in Figure 26 are:

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

System X

Program A	RR/SPM	CPI Communications	Database

System Y

Database	CPI Communications	RR/SPM	Program C

Programs A and C are in conversation

1 Receive (Program C → CPI Communications)

2 Set_Deallocate_Type (CM_DEALLOCATE_SYNC_LEVEL)

3 return_code=CM_OK

4 Deallocate

5 return_code=CM_OK

6 commit

Commit Protocols

7

8 data, return_code=CM_OK, *status_received*=CM_TAKE_COMMIT_DEALLOCATE

9 commit

10

11

12

13 return_code=ok return_code=ok

(the conversation is deallocated)

Figure 26. Conversation Deallocation Precedes the Commit Call

Part 2. CPI-C 2.1 Call Reference

Chapter 4. Call Reference 107
Call Syntax 108
Conformance Class and Interface Definition Table 109
Programming Language Considerations 111
  Application Generator 112
  C 112
  COBOL 112
  FORTRAN 112
  PL/I 112
  REXX 113
  RPG 113
How to Use the Call References 113
Summary List of Calls and Their Descriptions 114
Accept_Conversation (CMACCP) 119
Accept_Incoming (CMACCI) 121
Allocate (CMALLC) 124
Cancel_Conversation (CMCANC) 131
Confirm (CMCFM) 133
Confirmed (CMCFMD) 137
Convert_Incoming (CMCNVI) 139
Convert_Outgoing (CMCNVO) 141
Deallocate (CMDEAL) 143
Deferred_Deallocate (CMDFDE) 153
Extract_AE_Qualifier (CMEAEQ) 155
Extract_AP_Title (CMEAPT) 157
Extract_Application_Context_Name (CMEACN) 159
Extract_Conversation_Context (CMECTX) 161
Extract_Conversation_State (CMECS) 163
Extract_Conversation_Type (CMECT) 166
Extract_Initialization_Data (CMEID) 168
Extract_Mapped_Initialization_Data (CMEMID) 170
Extract_Maximum_Buffer_Size (CMEMBS) 173
Extract_Mode_Name (CMEMN) 175
Extract_Partner_ID (CMEPID) 177
Extract_Partner_LU_Name (CMEPLN) 180
Extract_Secondary_Information (CMESI) 182
Extract_Security_User_ID (CMESUI) 185
Extract_Send_Receive_Mode (CMESRM) 187
Extract_Sync_Level (CMESL) 189
Extract_TP_Name (CMETPN) 191
Extract_Transaction_Control (CMETC) 193
Flush (CMFLUS) 195
Include_Partner_In_Transaction (CMINCL) 198
Initialize_Conversation (CMINIT) 200
Initialize_For_Incoming (CMINIC) 203
Prepare (CMPREP) 205
Prepare_To_Receive (CMPTR) 208
Receive (CMRCV) 213
Receive_Expedited_Data (CMRCVX) 228
Receive_Mapped_Data (CMRCVM) 231

© Copyright IBM Corp. 1996, 1998

Release_Local_TP_Name (CMRLTP) 244
Request_To_Send (CMRTS) 246
Send_Data (CMSEND) 249
Send_Error (CMSERR) 259
Send_Expedited_Data (CMSNDX) 268
Send_Mapped_Data (CMSNDM) 271
Set_AE_Qualifier (CMSAEQ) 280
Set_Allocate_Confirm (CMSAC) 282
Set_AP_Title (CMSAPT) 284
Set_Application_Context_Name (CMSACN) 286
Set_Begin_Transaction (CMSBT) 288
Set_Confirmation_Urgency (CMSCU) 290
Set_Conversation_Security_Password (CMSCSP) 292
Set_Conversation_Security_Type (CMSCST) 295
Set_Conversation_Security_User_ID (CMSCSU) 298
Set_Conversation_Type (CMSCT) 301
Set_Deallocate_Type (CMSDT) 303
Set_Error_Direction (CMSED) 307
Set_Fill (CMSF) 310
Set_Initialization_Data (CMSID) 312
Set_Join_Transaction (CMSJT) 314
Set_Log_Data (CMSLD) 316
Set_Mapped_Initialization_Data (CMSMID) 318
Set_Mode_Name (CMSMN) 321
Set_Partner_ID (CMSPID) 323
Set_Partner_LU_Name (CMSPLN) 327
Set_Prepare_Data_Permitted (CMSPDP) 329
Set_Prepare_To_Receive_Type (CMSPTR) 331
Set_Processing_Mode (CMSPM) 334
Set_Queue_Callback_Function (CMSQCF) 337
Set_Queue_Processing_Mode (CMSQPM) 340
Set_Receive_Type (CMSRT) 344
Set_Return_Control (CMSRC) 346
Set_Send_Receive_Mode (CMSSRM) 349
Set_Send_Type (CMSST) 351
Set_Sync_Level (CMSSL) 354
Set_TP_Name (CMSTPN) 357
Set_Transaction_Control (CMSTC) 359
Specify_Local_TP_Name (CMSLTP) 361
Test_Request_To_Send_Received (CMTRTS) 363
Wait_For_Completion (CMWCMP) 366
Wait_For_Conversation (CMWAIT) 369

Chapter 4. Call Reference

This chapter describes the CPI Communications calls. Also, this chapter provides the function of each call and any optional setup calls that can be issued before the call being described. In addition, the following information is provided:

- **Communications Resource Manager Box**

  The CRM box identifies the CRMs on which the call is applicable. The possible values are LU 6.2 and OSI TP.

- **System Checklist Box**

  A system checklist precedes each call. If the call is implemented or announced on a particular system, that column is marked with an X. If it is not yet implemented on a particular system, that column is blank.

  Notes:

  1. The X does not mean all parameters are supported.
  2. If the call is not implemented, but an equivalent call is available as a product extension, the column is marked with an X*.

- **Format**

  The format used to program the call.

  Note: The actual syntax used to program the calls in this chapter depends on the programming language used. See “Call Syntax” on page 108 for specifics.

- **Parameters**

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

- **State Changes**

  The changes in the conversation state that can result from this call. See “Program Flow—States and Transitions” on page 52 for more information on conversation states.

- **Usage Notes**

  Additional information that applies to the call.

- **Related Information**

  Where to find additional information related to the call.

The CPI Communications interface definition is printed in black ink. If the implementation of an interface element in an operating environment differs from the CPI-C definition in its syntax or semantics, the text states that fact and is printed in green and has a *g* in the margin, as is this sentence.

Note: :Win32 means:

- IBM eNetwork Communication Server for Windows NT 5.0, 5.01, and above
- Win95 API Client for Communication Server 5.0, 5.01, and above
- WinNT API Client for Communication Server 5.0, 5.01, and above
- OS/2 API Client for Communication Server 5.0, 5.01, and above

© Copyright IBM Corp. 1996, 1998

- Win95 API Client for Netware for SAA 2.2
- WinNT API Client for Netware for SAA 2.2
- OS/2 API Client for Netware for SAA 2.2

- Win95 API Client for IntraNetware for SAA 2.3, 3.0 and above
- WinNT API Client for IntraNetware for SAA 2.3, 3.0 and above
- OS/2 API Client for IntraNetware for SAA 2.3, 3.0 and above

- IBM eNetwork Personal Communications 4.1 for WinNT and above
- IBM eNetwork Personal Communications 4.2 for Win95 and above

Call Syntax

CPI Communications calls can be made from application programs written in a number of high-level programming languages. Table 14 shows the languages that can be used on each IBM platform.

Table 14. Languages Supported by Platform

Language	AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
Application Generator		X	X	X		X	X	X	
C	X	X	X	X	X	X	X	X	X
COBOL		X	X	X	X	X	X	X	
FORTRAN			X	X		X	X	X	
PL/I		X	X	X				X	
REXX			X	X		X	X	X	
RPG			X	X			X		

In addition to the above languages, certain environments support CPI Communications calls in additional languages. For more information about supported languages as well as specific syntax and library information for each operating environment that implements CPI Communications, refer to the appropriate product chapters in “Part 3. CPI-C 2.1 Implementation Specifics” on page 373 for additional information.

This book uses a general call format to show the name of the CPI Communications call and the parameters used. This is an example of that format:

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

where `CMPROG` is the name of the call, and `parm0`, `parm1`, `parm2`, and `parmN` represent the parameter list described in the individual call description.

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

Application Generator (CSP)

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

C

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

COBOL

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

FORTRAN

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

PL/I

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

REXX

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

RPG

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

Conformance Class and Interface Definition Table

Table 15 lists the CPI Communications conformance classes, calls, and product implementation. More conformance class information is located in Chapter 5, "*CPI Communications 2.1 Conformance Classes*". An X is used to indicate which systems already have an IBM licensed program announced or available that implements a particular communications call. Refer to the footnotes for implementation specifics.

Table 15 (Page 1 of 3). CPI-C Calls and Product Implementation

Conformance Classes Calls	AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
Conversations (mandatory)									
Accept_Conversation	X	X	X	X	X	X	X	X	X
Allocate	X	X	X	X	X	X	X	X	X
Confirm	X	X	X	X	X	X	X	X	X
Confirmed	X	X	X	X	X	X	X	X	X
Deallocate	X	X	X	X	X	X	X	X	X
Extract_Conversation_State	X	X	X	X	X	X	X	X	X
Extract_Conversation_Type	X	X	X	X	X	X	X	X	X
Extract_Maximum_Buffer_Size	X[12]				X	X[1]	X		X
Extract_Mode_Name	X	X	X	X	X	X	X	X	X
Extract_Sync_Level	X	X	X	X	X	X	X	X	X
Flush	X	X	X	X	X	X	X	X	X
Initialize_Conversation	X	X	X	X	X	X	X	X	X
Prepare_To_Receive	X	X	X	X	X	X	X	X	X
Receive	X	X	X	X	X	X	X	X	X
Request_To_Send	X	X	X	X	X	X	X	X	X
Send_Data	X	X	X	X	X	X	X	X	X
Send_Error	X	X	X	X	X	X	X	X	X
Set_Conversation_Type	X	X	X	X	X	X	X	X	X
Set_Deallocate_Type	X	X	X	X	X	X	X	X	X
Set_Fill	X	X	X	X	X	X	X	X	X
Set_Log_Data	X	X	X	X	X	X	X	X	X
Set_Mode_Name	X	X	X	X	X	X	X	X	X
Set_Prepare_To_Receive_Type	X	X	X	X	X	X	X	X	X
Set_Receive_Type	X	X	X	X	X	X	X	X	X

Conformance Classes Calls	AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
Set_Return_Control	X	X	X	X	X	X	X	X	X
Set_Send_Type	X	X	X	X	X	X	X	X	X
Set_Sync_Level									
—CM_NONE[2]	X	X	X	X	X	X	X	X	X
—CM_CONFIRM[2]	X	X	X	X	X	X	X	X	X
Set_TP_Name	X	X	X	X	X	X	X	X	X
Test_Request_To_Send_Received	X	X	X	X	X	X	X	X	X
LU 6.2 (optional)									
Extract_Partner_LU_Name	X	X	X	X	X	X	X	X	X
Set_Error_Direction	X	X	X	X	X	X	X	X	X
Set_Partner_LU_Name	X	X	X	X	X	X	X	X	X
OSI TP (optional)									
Extract_AE_Qualifier									
Extract_AP_Title									
Extract_Application_Context_Name									
Set_Allocate_Confirm									
Set_AE_Qualifier									
Set_AP_Title									
Set_Application_Context_Name									
Set_Confirmation_Urgency									
Recoverable Transactions (optional)									
Set_Join_Transaction[3]									
Set_Sync_Level									
—CM_SYNC_POINT[2]		X					X	X	
—CM_SYNC_POINT_NO_CONFIRM[4]									
Deferred_Deallocate[5]									
Prepare[5]									
Set_Prepare_Data_Permitted[5]									
Unchained Transactions (optional)									
Extract_Transaction_Control									
Include_Partner_In_Transaction									
Set_Begin_Transaction									
Set_Transaction_Control									
Conversation-Level Non-Blocking (optional)									
Cancel_Conversation					X	X[8]			X
Set_Processing_Mode						X[8]			X
Wait_For_Conversation						X[8]			X
Queue-Level Non-Blocking (optional)									
Cancel_Conversation					X	X[8]			X
Set_Queue_Processing_Mode					X	X[8]			X
Wait_For_Completion					X	X[8]			X
Callback Function (optional)									
Cancel_Conversation					X				X
Set_Queue_Callback_Function					X	X[8]			X
Server (optional)									
Accept_Incoming	X[12]				X	X[8]			X
Extract_Conversation_Context	X[12]					X[8]			X
Extract_TP_Name	X[12]				X	X[8]			X
Initialize_For_Incoming	X[12]				X	X[8]			X
Release_Local_TP_Name	X[12]					X[8]			X
Specify_Local_TP_Name	X[12]					X[8]			X
Data Conversion Routines (optional)									
Convert_Incoming	X[12].				X	X[8]	X		X
Convert_Outgoing	X[12]				X	X[8]	X		X
Security (optional)									
Extract_Security_User_ID	X[11]				X	X[9]	X	X[6]	X
Set_Conversation_Security_Password	X[11]				X	X[9]	X	X[6]	X
Set_Conversation_Security_Type									
—CM_SECURITY_NONE[7]	X[11]				X	X[9]	X	X[6]	X
—CM_SECURITY_PROGRAM[7]	X[11]				X	X[9]	X	X[6]	X
—CM_SECURITY_PROGRAM_ STRONG[7]							X		
—CM_SECURITY_SAME[7]	X[11]				X	X[9]	X	X[6]	X
Set_Conversation_Security_User_ID	X[11]				X	X[9]	X	X[6]	X
Distributed Security (optional)									
Extract_Security_User_ID	X[11]				X	X[9]	X	X[6]	X
Set_Conversation_Security_Type									
—CM_SECURITY_NONE[7]	X[11]				X	X[9]	X	X[6]	X
—CM_SECURITY_SAME[7]	X[11]				X	X[9]	X	X[6]	X
—CM_SECURITY_DISTRIBUTED[7]									
—CM_SECURITY_MUTUAL[7]									

Table 15 (Page 2 of 3). CPI-C Calls and Product Implementation

Table 15 (Page 3 of 3). CPI-C Calls and Product Implementation

Conformance Classes / Calls	AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
Full-Duplex (optional)									
Extract_Send_Receive_Mode					X	X[8]			X[14]
Set_Send_Receive_Mode					X	X[8]			X[14]
Expedited Data (optional)									
Receive_Expedited_Data						X[8]			X[14]
Send_Expedited_Data						X[8]			X[14]
Directory (optional)									
Extract_Partner_ID									
Set_Partner_ID									
Secondary Information (optional)									
Extract_Secondary_Information						X[10]			X[14]
Initialization Data (optional)									
Extract_Initialization_Data									
Set_Initialization_Data									
Automatic Data Conversion									
Extract_Mapped_Initialization_Data[13]									
Receive_Mapped_Data									
Send_Mapped_Data									
Set_Mapped_Initialization_Data[13]									

Notes:

[1] This call is supported in Communications Manager/2 Version 1.11 and IBM Communications Server.
[2] This is a required *sync_level* value.
[3] This is a required call if the TX resource recovery interface is supported.
[4] This is a required *sync_level* value if either full duplex or OSI TP is also supported.
[5] This is a required call if OSI TP is also supported.
[6] This platform currently supports the function provided by this call via a product extension call. Refer to the appropriate product chapter.
[7] This is a required *conversation_security_type* value.
[8] This call is supported only in IBM Communications Server, and only at the 32-bit C Language interface and through REXX.
[9] This call is supported only in IBM Communications Server, and only at the 32-bit C Language interface and through REXX. In Communications Manager (prior to IBM Communications Server) this call is supported via a product extension call. Refer to the OS/2 chapter for details.
[10] This call is supported only in IBM Communications Server, and only at the 32-bit C Language interface and through REXX.
[11] This call is supported in AIX SNA Server Version 3 Release 1 or later. In SNA Server, prior to Version 3, this call is supported via a product extension call. Refer to the AIX chapter for details.
[12] This call is supported in AIX SNA Server Version 3 Release 1 or later.
[13] This is a required call if Initialization Data is also supported.
[14] This call is not supported for the following Win32 CPI-C platforms:

- Win95 API Client for Communication Server 5.0, 5.01, and above
- WinNT API Client for Communication Server 5.0, 5.01, and above
- OS/2 API Client for Communication Server 5.0, 5.01, and above
- Win95 API Client for Netware for SAA 2.2
- WinNT API Client for Netware for SAA 2.2
- OS/2 API Client for Netware for SAA 2.2
- Win95 API Client for IntraNetware for SAA 2.3, 3.0 and above
- WinNT API Client for IntraNetware for SAA 2.3, 3.0 and above
- OS/2 API Client for IntraNetware for SAA 2.3, 3.0 and above

Programming Language Considerations

This section describes programming language considerations to keep in mind when writing and running a program that uses CPI Communications. Sample pseudonym files are on the diskette that came with this manual. Customized pseudonym files or datasets for supported programming languages may be available on systems that implement CPI Communications.

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

Application Generator

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

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

C

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

- When passing an integer value as a parameter, prefix the parameter name with an ampersand (&) so that the value is passed by reference.
- To pass a parameter as a string literal, surround it with double quotes rather than single quotes.
- To enable asynchronous updates of program variables, the return parameters on a non-blocking call must be declared using the `volatile` qualifier as defined in ANSI C.

COBOL

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

- Because COBOL does not support the underscore character (_), the underscores in COBOL pseudonyms are replaced with dashes (-). For example, COBOL programmers use CM-IMMEDIATE as a pseudonym value name in their programs instead of CM_IMMEDIATE.
- Each argument in the parameter list must be called (listed) by name.
- Each variable in the parameter list must be level 01.
- Number variables must be full words (at least five but less than ten "9"s) and they must be COMP-4, not zoned decimal.

FORTRAN

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

- The EXTERNAL statement may be required for each CPI Communications call that is issued, depending on the environment being used. The PRAGMA statement may also be required. EXTERNAL and PRAGMA statements may be included in the FORTRAN pseudonym file provided for a given environment.
- To enable asynchronous updates of program variables, the return parameters on a non-blocking call must be declared using the `VOLATILE` qualifier.

PL/I

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

- Numbers in the parameter list must be declared, initialized, and passed as variables.
- ENTRY declaration statements should be used for each CPI Communications call that is issued. ENTRY declaration statements may be included in the PL/I pseudonym file provided for a given environment.
- To enable asynchronous updates of program variables, the return parameters on a non-blocking call must be declared using the `ABNORMAL` qualifier.

REXX

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

- REXX programs must use the ADDRESS CPICOMM statement to access CPI Communications calls. These calls are not accessible through the REXX CALL statement interface.
- Character strings returned by CPI Communications calls are stored in variables with the maximum allowable length. (Maximum lengths are shown in Appendix A, "Variables and Characteristics.") However, there is a returned length variable associated with the returned character string, which allows use of the REXX function

  ```
  LEFT(returned_char_string,returned_length)
  ```

  to get the correct amount of data.

  (This note does not apply to returned fixed-length character strings. For instance, a *conversation_ID* returned from the Accept_Conversation call always has a length of 8 bytes.)

RPG

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

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

How to Use the Call References

Here is an example of how the information in this chapter can be used in connection with the rest of the book. The example describes how to use the Set_Return_Control call to set the conversation characteristic of *return_control* to a value of CM_IMMEDIATE.

- Table 15 on page 109 shows that Set_Return_Control is implemented for all operating environments.
- "Set_Return_Control (CMSRC)" on page 346 contains the semantics of the variables used for the call. It explains that the real name of the program call for Set_Return_Control is CMSRC and that CMSRC has a parameter list of *conversation_ID*, *return_control*, and *return_code*.
- "Call Syntax" on page 108 shows the syntax for the programming language being used.
- Appendix A, "Variables and Characteristics" provides a complete description of all variables used in the book and shows that the *return_control* variable, which goes into the call as a parameter, is a 32-bit integer. This information is provided in Table 61 on page 650.
- Table 59 on page 642 in Appendix A, "Variables and Characteristics" shows that CM_IMMEDIATE, which is placed into the *return_control* parameter on the call to CMSRC, is defined as having an integer value of 1.
- Finally, the *return_code* value CM_OK, which is returned to the program on the CMSRC call, is defined in Appendix B, "Return Codes and Secondary Information." CM_OK means that the call completed successfully.

Summary List of Calls and Their Descriptions

Table 16 (Page 1 of 5). List of CPI-C Calls and Their Descriptions

Pseudonym	Call	Description	Page
Accept_Conversation	CMACCP	Used by a program to accept an incoming conversation.	119
Accept_Incoming	CMACCI	Used by a program to accept an incoming conversation previously initialized with the Initialize_For_Incoming call.	121
Allocate	CMALLC	Used by a program to establish a conversation.	124
Cancel_Conversation	CMCANC	Used by a program to end a conversation immediately.	131
Confirm	CMCFM	Used by a program to send a confirmation request to its partner.	133
Confirmed	CMCFMD	Used by a program to send a confirmation reply to its partner.	137
Convert_Incoming	CMCNVI	Used by a program to change the encoding of a character string from EBCDIC to the local encoding used by the program.	139
Convert_Outgoing	CMCNVO	Used by a program to change the encoding of a character string from the local encoding used by the program to EBCDIC.	141
Deallocate	CMDEAL	Used by a program to end a conversation.	143
Deferred_Deallocate	CMDFDE	Used by a program to end a conversation following successful completion of the current transaction.	153
Extract_AE_Qualifier	CMEAEQ	Used by a program to view the current *ae_qualifier* conversation characteristic.	155
Extract_AP_Title	CMEAPT	Used by a program to view the current *ap_title* conversation characteristic.	157
Extract_Application_Context_Name	CMEACN	Used by a program to view the current *application_context_name* conversation characteristic.	159
Extract_Conversation_Context	CMECTX	Used by a program to extract the *context_ID* for a conversation.	161
Extract_Conversation_State	CMECS	Used by a program to view the current state of a conversation.	163
Extract_Conversation_Type	CMECT	Used by a program to view the current *conversation_type* conversation characteristic.	166
Extract_Initialization_Data	CMEID	Used by a program to extract the current *initialization_data* conversation characteristic.	168
Extract_Mapped_Initialization_Data	CMEMID	Used by a program to extract the *initialization_data* conversation characteristic for mapped initialization data.	170
Extract_Maximum_Buffer_Size	CMEMBS	Used by a program to extract the maximum buffer size supported by the system.	173

Table 16 (Page 2 of 5). List of CPI-C Calls and Their Descriptions

Pseudonym	Call	Description	Page
Extract_Mode_Name	CMEMN	Used by a program to view the current *mode_name* conversation characteristic.	175
Extract_Partner_ID	CMEPID	Used by a program to view the current *partner_ID* conversation characteristic.	177
Extract_Partner_LU_Name	CMEPLN	Used by a program to view the current *partner_LU_name* conversation characteristic.	180
Extract_Secondary_Information	CMESI	Used by a program to extract secondary information associated with the return code for a given call.	182
Extract_Security_User_ID	CMESUI	Used by a program to view the current *security_user_ID* conversation characteristic.	185
Extract_Send_Receive_Mode	CMESRM	Used by a program to view the current *send_receive_mode* conversation characteristic.	187
Extract_Sync_Level	CMESL	Used by a program to view the current *sync_level* conversation characteristic.	189
Extract_TP_Name	CMETPN	Used by a program to determine the *TP_name* characteristic's value for a given conversation.	191
Extract_Transaction_Control	CMETC	Used by a program to extract the *transaction_control* characteristic's value for a given conversation.	193
Flush	CMFLUS	Used by a program to flush the local CRM's send buffer.	195
Include_Partner_In_Transaction	CMINCL	Used by a program to include a partner program in a transaction.	198
Initialize_Conversation	CMINIT	Used by a program to initialize the conversation characteristics for an outgoing conversation.	200
Initialize_For_Incoming	CMINIC	Used by a program to initialize the conversation characteristics for an incoming conversation.	203
Prepare	CMPREP	Used by a program to prepare a subordinate for a commit operation.	205
Prepare_To_Receive	CMPTR	Used by a program to change a conversation from **Send** to **Receive** state in preparation to receive data.	208
Receive	CMRCV	Used by a program to receive data.	213
Receive_Expedited_Data	CMRCVX	Used by a program to receive expedited data from its partner.	228
Receive_Mapped_Data	CMRCVM	Used by a program to receive mapped data from its partner.	231
Release_Local_TP_Name	CMRLTP	Used by a program to release a name.	244
Request_To_Send	CMRTS	Used by a program to notify its partner that it would like to send data.	246
Send_Data	CMSEND	Used by a program to send data.	249

Table 16 (Page 3 of 5). List of CPI-C Calls and Their Descriptions

Pseudonym	Call	Description	Page
Send_Error	CMSERR	Used by a program to notify its partner of an error that occurred during the conversation.	259
Send_Expedited_Data	CMSNDX	Used by a program to send expedited data to its partner.	268
Send_Mapped_Data	CMSNDM	Used by a program to send mapped data to its partner	271
Set_AE_Qualifier	CMSAEQ	Used by a program to set the *ae_qualifier* conversation characteristic.	280
Set_Allocate_Confirm	CMSAC	Used by a program to set the *allocate_confirm* conversation characteristic.	282
Set_AP_Title	CMSAPT	Used by a program to set the *ap_title* conversation characteristic.	284
Set_Application_Context_Name	CMSACN	Used by a program to set the *application_context_name* conversation characteristic.	286
Set_Begin_Transaction	CMSBT	Used by a program to set the *begin_transaction* conversation characteristic.	288
Set_Confirmation_Urgency	CMSCU	Used by a program to set the *confirmation_urgency* conversation characteristic.	286
Set_Conversation_Security_Password	CMSCSP	Used by a program to set the *security_password* conversation characteristic.	292
Set_Conversation_Security_Type	CMSCST	Used by a program to set the *conversation_security_type* conversation characteristic.	295
Set_Conversation_Security_User_ID	CMSCSU	Used by a program to set the *security_user_ID* conversation characteristic.	298
Set _Conversation_Type	CMSCT	Used by a program to set the *conversation_type* conversation characteristic.	301
Set_Deallocate_Type	CMSDT	Used by a program to set the *deallocate_type* conversation characteristic.	303
Set_Error_Direction	CMSED	Used by a program to set the *error_direction* conversation characteristic.	307
Set_Fill	CMSF	Used by a program to set the *fill* conversation characteristic.	310
Set_Initialization_Data	CMSID	Used by a program to set the *initialization_data* conversation characteristic.	312
Set_Join_Transaction	CMSJT	Used by a program to set the *join_transaction* conversation characteristic.	314
Set_Log_Data	CMSLD	Used by a program to set the *log_data* conversation characteristic.	316

Table 16 (Page 4 of 5). List of CPI-C Calls and Their Descriptions

Pseudonym	Call	Description	Page
Set_Mapped_Initialization_Data	CMSMID	Used by a program to set the initialization_data conversation characteristic for mapped initialization data.	318
Set_Mode_Name	CMSMN	Used by a program to set the *mode_name* conversation characteristic.	321
Set_Partner_ID	CMSPID	Used by a program to set the *partner_ID* conversation characteristic.	323
Set_Partner_LU_Name	CMSPLN	Used by a program to set the *partner_LU_name* conversation characteristic.	327
Set_Prepare_Data_Permitted	CMSPDP	Used by a program to set the *prepare_data_permitted* conversation characteristic.	329
Set_Prepare_To_Receive_Type	CMSPTR	Used by a program to set the *prepare_to_receive_type* conversation characteristic.	331
Set_Processing_Mode	CMSPM	Used by a program to set the *processing_mode* conversation characteristic.	334
Set_Queue_Callback_Function	CMSQCF	Used by a program to set a callback function and a user field for a given conversation queue and to set the queue's processing mode to CM_NON_BLOCKING.	337
Set_Queue_Processing_Mode	CMSQPM	Used by a program to set the processing mode for a given conversation queue and to associate an outstanding-operation identifier (OOID) and a user field with the queue.	340
Set_Receive_Type	CMSRT	Used by a program to set the *receive_type* conversation characteristic.	344
Set_Return_Control	CMSRC	Used by a program to set the *return_control* conversation characteristic.	346
Set_Send_Receive_Mode	CMSSRM	Used by a program to set the *send_receive_mode* conversation characteristic.	349
Set_Send_Type	CMSST	Used by a program to set the *send_type* conversation characteristic.	351
Set_Sync_Level	CMSSL	Used by a program to set the *sync_level* conversation characteristic.	354
Set_TP_Name	CMSTPN	Used by a program to set the *TP_Name* conversation characteristic.	357
Set_Transaction_Control	CMSTC	Used by a program to set the *transaction_control* conversation characteristic.	359
Specify_Local_TP_Name	CMSLTP	Used by a program to associate a name with itself.	361
Test_Request_To_Send_Received	CMTRTS	Used by a program to determine whether or not the remote program is requesting to send data.	363

Table 16 (Page 5 of 5). List of CPI-C Calls and Their Descriptions

Pseudonym	Call	Description	Page
Wait_For_Completion	CMWCMP	Used by a program to wait for completion of one or more outstanding operations represented in a specified outstanding-operation-ID (OOID) list.	366
Wait_For_Conversation	CMWAIT	Used by a program to wait for the completion of any conversation-level outstanding operation.	369

Accept_Conversation (CMACCP)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

The Accept_Conversation (CMACCP) call accepts an incoming conversation. Like Initialize_Conversation, this call initializes values for various conversation characteristics. The difference between the two calls is that the program that will later allocate the conversation issues the Initialize_Conversation call, and the partner program that will accept the conversation after it is allocated issues the Accept_Conversation call.

Format

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

Parameters

conversation_ID *(output)*
Specifies the conversation identifier assigned to the conversation. CPI Communications supplies and maintains the *conversation_ID*. When the *return_code* is set equal to CM_OK, the value returned in this parameter is used by the program on all subsequent calls issued for this conversation.

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

- CM_OK
- CM_DEALLOCATED_ABEND
  This value indicates, that CPI Communications deallocated the incoming conversation because an implicit call of tx_set_transaction_control or tx_begin failed.
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  – No incoming conversation exists.
  – No name is associated with the program. A program associates a name with itself by issuing the Specify_Local_TP_Name call.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

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

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

Usage Notes

1. For each conversation, CPI Communications assigns a unique identifier (the *conversation_ID*) that the program uses in all future calls intended for that conversation. Therefore, the program must issue the Accept_Conversation call before any other calls can refer to the conversation.
2. There may be a system-defined limit on the number of conversations that a program can accept or allocate, but CPI Communications imposes no limit.
3. For a list of the conversation characteristics that are initialized when the Accept_Conversation call completes successfully, see Table 3 on page 35.
4. CPI Communications makes incoming conversations available to programs based upon names that are associated with the program. Specifically, those names associated with the program at the time the Accept_Conversation call is issued are used to satisfy that Accept_Conversation call. These names come either from locally defined information or from execution of the Specify_Local_TP_Name call. An implementation may place restrictions on the actions that a program may take before issuing Accept_Conversation in order to properly identify programs with associated names.
5. An implementation may choose to specify a minimum time before returning CM_PROGRAM_STATE_CHECK when no incoming conversation has arrived for the program.
6. Accept_Conversation always functions as if the *processing_mode* were set to CM_BLOCKING. A program that must be able to accept incoming conversations in a non-blocking mode should use the Initialize_For_Incoming and Accept_Incoming calls. The processing mode for the conversation can be set to CM_NON_BLOCKING prior to issuing Accept_Incoming.
7. A new context is created as a result of the successful completion of the Accept_Conversation call, and the conversation is assigned to the new context. The program's current context is set to the new context by node services.

Related Information

"Conversation Characteristics" on page 33 provides a comparison of the conversation characteristics set by Initialize_For_Incoming, Initialize_Conversation, and Accept_Conversation.

"Example 1: Data Flow in One Direction" on page 69 shows an example program flow using the Accept_Conversation call to accept a half-duplex conversation.

"Example 8: Establishing a Full-Duplex Conversation" on page 84 shows an example program flow using an Accept_Conversation call to accept a full-duplex conversation.

"Initialize_Conversation (CMINIT)" on page 200 describes how the conversation characteristics are initialized for the program that allocates the conversation.

Accept_Incoming (CMACCI)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X*				X	X*			X

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

Before issuing the Accept_Incoming call, a program has the option of issuing one of the following calls:

CALL CMSPM – Set_Processing_Mode
CALL CMSQPM – Set_Queue_Processing_Mode
CALL CMSQCF – Set_Queue_Callback_Function

X* In AIX, this call is supported in Version 3 Release 1 or later. In OS/2, this call is supported by Communications Server.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier of a conversation that has been initialized for an incoming conversation.

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_DEALLOCATED_ABEND
  This value indicates, that CPI Communications deallocated the incoming conversation because an implicit call of tx_set_transaction_control or tx_begin failed.
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  – The conversation is not in **Initialize-Incoming** state.
  – No name is associated with the program. A program associates a name with itself by issuing the Specify_Local_TP_Name call.

- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

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

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

Usage Notes

1. The Accept_Incoming call can be used only when an Initialize_For_Incoming call has already completed.
2. When Accept_Incoming successfully completes, CPI Communications initializes those conversation characteristics that use values from the conversation startup request. See Table 3 on page 35 for a list of the conversation characteristics and how they are set by Initialize_For_Incoming and Accept_Incoming.
3. If Accept_Incoming is issued as a blocking call and no incoming conversation is available for the program, the call blocks until a conversation startup request arrives. The program can ensure that it is not placed in a wait state by taking one of the following actions before issuing the Accept_Incoming call:
   - For conversation-level non-blocking — set the *processing_mode* characteristic to CM_NON_BLOCKING by using the Set_Processing_Mode call
   - For queue-level non-blocking — set the processing mode for the Initialization queue to CM_NON_BLOCKING by using the Set_Queue_Callback_Function call or the Set_Queue_Processing_Mode call.
4. If the program has successfully issued a Set_Processing_Mode call, the subsequent Accept_Incoming call will complete only when the conversation startup request is for a half-duplex conversation.
5. There may be a system-defined limit on the number of conversations that a program can accept or allocate, but CPI Communications imposes no limit.
6. CPI Communications makes incoming conversations available to programs based upon names that are associated with the program. Specifically, those names associated with the program at the time the Accept_Incoming call is issued are used to satisfy that Accept_Incoming call. These names come either from locally defined information or from execution of the Specify_Local_TP_Name call. An implementation may place restrictions on the actions that a program may take before issuing Accept_Incoming in order to properly identify programs with associated names.
7. A new context is created as a result of the successful completion of the Accept_Incoming call, and the conversation is assigned to the new context. If Accept_Incoming completes successfully with *return_code* set to CM_OK, the program's current context is set to the new context by node services. If Accept_Incoming is issued with *processing_mode* set to CM_NON_BLOCKING and gets the CM_OPERATION_INCOMPLETE return code, the program's current context is not changed if the Accept_Incoming call operation subsequently completes successfully as a result of the Wait_For_Conversation call.

Related Information

“Conversation Characteristics” on page 33 provides a comparison of the conversation characteristics set by Initialize_For_Incoming and Accept_Incoming.

“Example 12: Accepting Multiple Conversations Using Blocking Calls” on page 92 and “Example 13: Accepting Multiple Conversations Using Conversation-Level Non-Blocking Calls” on page 94 show example program flows using the Initialize_For_Incoming and Accept_Incoming calls.

“Initialize_For_Incoming (CMINIC)” on page 203 describes how the *conversation_ID* supplied on Accept_Incoming is assigned.

“Set_Processing_Mode (CMSPM)” on page 334 describes setting the *processing mode* conversation characteristic.

“Set_Queue_Callback_Function (CMSQCF)” on page 337 describes the how to set a callback function and related information for a non-blocking conversation queue.

“Set_Queue_Processing_Mode (CMSQPM)” on page 340 describes how to set the processing mode for a non-blocking conversation queue.

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

Allocate (CMALLC)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

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

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

CALL CMSAEQ – Set_AE_Qualifier
CALL CMSAC – Set_Allocate_Confirm
CALL CMSAPT – Set_AP_Title
CALL CMSACN – Set_Application_Context_Name
CALL CMSBT – Set_Begin_Transaction
CALL CMSCSP – Set_Conversation_Security_Password
CALL CMSCST – Set_Conversation_Security_Type
CALL CMSCSU – Set_Conversation_Security_User_ID
CALL CMSCT – Set_Conversation_Type
CALL CMSID – Set_Initialization_Data
CALL CMSMID – Set_Mapped_Initialization_Data
CALL CMSMN – Set_Mode_Name
CALL CMSPLN – Set_Partner_LU_Name
CALL CMSPM – Set_Processing_Mode
CALL CMSQCF – Set_Queue_Callback_Function
CALL CMSQPM – Set_Queue_Processing_Mode
CALL CMSRC – Set_Return_Control
CALL CMSSRM – Set_Send_Receive_Mode
CALL CMSSL – Set_Sync_Level
CALL CMSTPN – Set_TP_Name
CALL CMSTC – Set_Transaction_Control

Format

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

Parameters

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

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

- CM_OK
- CM_OPERATION_INCOMPLETE

- CM_CONVERSATION_CANCELLED
- CM_RETRY_LIMIT_EXCEEDED
  This value indicates that the system-specified retry limit was exceeded.
- CM_SEND_RCV_MODE_NOT_SUPPORTED
- CM_SYNC_LVL_NOT_SUPPORTED_SYS
- CM_PARAMETER_ERROR
  This value indicates one of the following:
  – The *mode_name* characteristic (set from side information or by Set_Mode_Name) specifies a mode name that is not recognized by the LU as being valid.
  – The *mode_name* characteristic (set from side information or by Set_Mode_Name) specifies a mode name that the local program does not have the authority to specify. For example, SNASVCMG requires special authority with LU 6.2.
  – The *TP_name* characteristic (set from side information or by Set_TP_Name) specifies a transaction program name that the local program does not have the appropriate authority to allocate a conversation to. For example, SNA service programs require special authority with LU 6.2. (For more information, see "SNA Service Transaction Programs" on page 727.)
  – The *TP_name* characteristic (set from side information or by Set_TP_Name) specifies an SNA service transaction program and *conversation_type* is set to CM_MAPPED_CONVERSATION.
  – The *partner_LU_name* characteristic (set from side information or by Set_Partner_LU_Name) specifies a partner LU name that is not recognized as being valid.
  – The *AP_title* characteristic (set from side information or using the Set_AP_Title call) or the *AE_qualifier* characteristic (set from side information or using the Set_AE_Qualifier call), or the *application_context_name* characteristic (set from side information or using the Set_Application_Context_Name call) specifies an AP title or an AE qualifier or an application context name that is not recognized as being valid.
  – The *conversation_security_type* characteristic is CM_SECURITY_PROGRAM or CM_SECURITY_PROGRAM_STRONG, and the *security_password* characteristic or the *security_user_ID* characteristic (set from side information or by SET calls), or both, are null.
  – The *conversation_security_type* is set to CM_SECURITY_DISTRIBUTED or CM_SECURITY_MUTUAL and the partner principal name (set from the program binding) is null or is not recognized by the CRM as being valid.
  – A *partner_ID* characteristic was provided that caused a search of the distributed directory, but no program binding was retrieved.
  – The program binding for the conversation, either specified directly on the Set_Partner_ID call or obtained from the distributed directory, was invalid.
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  – The conversation is not in **Initialize** state.
  – The *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, *transaction_control* is set to CM_CHAINED_TRANSACTIONS, and the conversation's context is not in transaction.

  - For a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the conversation's context is in the **Backout-Required** condition. New protected conversations cannot be allocated for a context when it is in this condition.
  - The program has issued a successful Accept_Conversation or Accept_Incoming call on a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and using an OSI TP CRM, and the program has not issued a Receive call on this conversation.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR
- CM_SECURITY_NOT_SUPPORTED
  This value indicates that the requested conversation security type could not be provided. This is either because the remote system does not accept the requested type of security from the local system or because the requested security does not transport the type of required user name identified in the program binding.

If *conversation_security_type* is set to CM_SECURITY_MUTUAL, *return_code* can have the following values:

- CM_SECURITY_NOT_VALID
  This value indicates that the remote system rejected the conversation startup request due to a security-related error.
- CM_SECURITY_MUTUAL_FAILED
  This value indicates that the remote system could not be successfully authenticated.
- CM_CONVERSATION_TYPE_MISMATCH
- CM_PIP_NOT_SPECIFIED_CORRECTLY
- CM_SYNC_LVL_NOT_SUPPORTED_PGM
- CM_TPN_NOT_RECOGNIZED
- CM_TP_NOT_AVAILABLE_NO_RETRY
- CM_TP_NOT_AVAILABLE_RETRY
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY

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

- CM_ALLOCATE_FAILURE_NO_RETRY
- CM_ALLOCATE_FAILURE_RETRY

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

- CM_UNSUCCESSFUL

  This value indicates that the logical connection is not immediately available.

State Changes

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

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

Usage Notes

1. An allocation error resulting from the local system's failure to obtain a logical connection for the conversation is reported on the Allocate call. An allocation error resulting from the remote system's rejection of the conversation startup request is reported on a subsequent conversation call, unless *conversation_security_type* is set to CM_SECURITY_MUTUAL. In this case, the allocation error is reported on the Allocate call.
2. For CPI Communications to establish the conversation, CPI Communications must first establish a logical connection between the local system and the remote system, if such a connection does not already exist.
3. Depending on the circumstances, the local system can send the conversation startup request to the remote system as soon as it allocates a logical connection for the conversation. The local system can also buffer the conversation startup request until it accumulates enough information for transmission (from one or more subsequent Send_Data calls), or until the local program issues a subsequent call other than Send_Data that explicitly causes the system to flush its send buffer. The amount of information sufficient for transmission depends on the characteristics of the logical connection allocated for the conversation and can vary from one logical connection to another.
4. The local program can ensure that the remote program is connected as soon as possible by issuing Flush (CMFLUS) immediately after Allocate (CMALLC).
5. A set of security parameters is established for the conversation, based on the values of the security characteristics. See “Conversation Security” on page 51 for more information on conversation security.
6. When *return_control* is set to CM_IMMEDIATE, the call completes immediately, regardless of the processing mode in effect for the Allocate call. If a logical connection is not available, *return_code* is set to CM_UNSUCCESSFUL.
7. When a program allocates a conversation, the program's current context at the time the Allocate call is issued becomes the new conversation's context. If the program has multiple contexts (for example, as a result of accepting multiple conversations), it must ensure that the current context is set to the appropriate context before allocating the new conversation.
8. Initialization data specified by use of the Set_Initialization_Data (CMSID) call is sent to the remote program along with the conversation startup request. The remote program may extract the initialization data with the Extract_Initialization_Data (CMEID) call.
9. Initialization data specified by the use of the Set_Mapped_Initialization_Data (CMSMID) call is sent to the remote program along with the conversation startup request. The remote program may extract the initialization data with the Extract_Mapped_Initialization_Data (CMEMID) call.

10. By using the Set_Allocate_Confirm call, the program allocating the conversation may request notification that the remote program has confirmed its acceptance of the conversation.

11. If a conversation is using a particular CRM type, the Allocate call tries to establish a conversation using only the destination information for that CRM type.

12. If a program specifies destination information for both an OSI TP CRM and an LU 6.2 CRM but only one set of information is complete, the Allocate call tries only the destination for which CPI Communications has complete information. If complete destination information exists for use of both an LU 6.2 CRM and an OSI TP CRM, the Allocate call tries to establish a logical connection using one and then the other destination. Only if both attempts fail does the Allocate call return either CM_ALLOCATE_FAILURE_* or CM_UNSUCCESSFUL.

13. CPI Communications programs can provide multiple program bindings for use on Allocate:

    - The program specifies a *distinguished_name* (either in side information or using the Set_Partner_ID call) that identifies a directory object containing multiple program bindings.
    - The program specifies a PFID with the Set_Partner_ID call that identifies multiple program installation objects with, possibly, multiple program bindings.
    - The program issues the Set_Partner_ID call with *partner_ID_scope* set to CM_REFERENCE. This allows CPI Communications to locate alternative installations of the same function if unable to establish a logical connection using the program binding obtained from the specified DN.
    - A program may use the Set_Partner_ID call with *partner_ID_type* set to CM_PROGRAM_BINDING and provide a *partner_ID* characteristic that contains multiple program bindings.

    In each of these cases, CPI Communications uses all available bindings to attempt to establish a logical connection, retrying until a logical connection is established or all bindings have been tried. The order of bindings used is determined by the method used. For example, a CM_REFERENCE option implies an ordering; the DN specified should be tried first. Alternatively, some implementations may be able to determine a “least cost” or “closest” partner and create an ordering based on network topology. Where no ordering for the bindings exists, random selection is enforced.

    Although no retry limit is provided at the CPI Communications level, a retry limit may be imposed by the local system to control the effects of retry. The form of this limit is system-specific; however, if a limit exists and is reached, the program is informed of this condition with a return code of CM_RETRY_LIMIT_EXCEEDED on the Allocate.

14. When *conversation_security_type* is CM_SECURITY_DISTRIBUTED or CM_SECURITY_MUTUAL, then the partner principal name must be set via a program binding before the Allocate. This can be done explicitly using the CMSPID call with *partner_ID_type* set to CM_PROGRAM_BINDING or implicitly using a directory.

15. Various *conversation_security_type* values and required_user_name_type values (from the program binding) cause the local CRM to reject an Allocate

request with a *return_code* of CM_SECURITY_NOT_SUPPORTED. The incompatible combinations are shown in Table 8 on page 52.

CPI Communications applications in CICS cannot be SNA service programs and, therefore, cannot allocate on the mode names SNASVCMG or CPSVCMG. If they attempt to do this, they get the CM_PARAMETER_ERROR return code.

Related Information

"Contexts and Context Management" on page 32 defines contexts and discusses how application programs can manage multiple contexts.

"Example 1: Data Flow in One Direction" on page 69 shows an example program flow using the Allocate call to establish a half-duplex conversation.

"Data Buffering and Transmission" on page 44 discusses control methods for data transmission.

"Example 8: Establishing a Full-Duplex Conversation" on page 84 shows an example program flow using an Allocate call to establish a full-duplex conversation.

"Extract_Conversation_Context (CMECTX)" on page 161 discusses the *context_ID* characteristic.

"Extract_Partner_ID (CMEPID)" on page 177 discusses the *partner_ID* characteristic.

"Set_AE_Qualifier (CMSAEQ)" on page 280 discusses the *AE_qualifier* conversation characteristic.

"Set_Allocate_Confirm (CMSAC)" on page 282 discusses the *allocate_confirm* conversation characteristic and explains an option for confirming acceptance of the conversation.

"Set_AP_Title (CMSAPT)" on page 284 discusses the *AP_title* conversation characteristic.

"Set_Application_Context_Name (CMSACN)" on page 286 discusses the *application_context_name* conversation characteristic.

"Set_Begin_Transaction (CMSBT)" on page 288 discusses the *begin_transaction* conversation characteristic.

"Set_Conversation_Security_Password (CMSCSP)" on page 292 discusses the *security_password* conversation characteristic.

"Set_Conversation_Security_Type (CMSCST)" on page 295 discusses the *conversation_security_type* conversation characteristic.

"Set_Conversation_Security_User_ID (CMSCSU)" on page 298 discusses the *security_user_ID* conversation characteristic.

"Set_Conversation_Type (CMSCT)" on page 301 discusses the *conversation_type* characteristic.

“Set_Initialization_Data (CMSID)” on page 312 and “Extract_Initialization_Data (CMEID)” on page 168 discuss the *initialization_data* conversation characteristic.

“Set_Mode_Name (CMSMN)” on page 321 discusses the *mode_name* conversation characteristic.

“Set_Partner_ID (CMSPID)” on page 323 discusses the *partner_ID* characteristic.

“Set_Mapped_Initialization_Data (CMSMID)” on page 318 discusses the *initialization_data conversation* characteristic for mapped initialization data.

“Set_Partner_LU_Name (CMSPLN)” on page 327 discusses the *partner_LU_name* conversation characteristic.

“Set_Processing_Mode (CMSPM)” on page 334 describes setting the *processing_mode* conversation characteristic.

“Set_Queue_Callback_Function (CMSQCF)” on page 337 discusses how to set a callback function and related information for a conversation queue.

“Set_Queue_Processing_Mode (CMSQPM)” on page 340 discusses how to set the processing mode for a conversation queue.

“Set_Return_Control (CMSRC)” on page 346 discusses the *return_control* characteristic.

“Set_Send_Receive_Mode (CMSSRM)” on page 349 discusses how to set the send-receive mode for a conversation.

“Set_Sync_Level (CMSSL)” on page 354 discusses the *sync_level* conversation characteristic.

“Set_TP_Name (CMSTPN)” on page 357 discusses the *TP_name* conversation characteristic.

“Set_Transaction_Control (CMSTC)” on page 359 discusses the *transaction_control* conversation characteristic.

“Program Binding” on page 658 describes the format of a program binding.

“SNA Service Transaction Programs” on page 727 discusses SNA service transaction programs.

Cancel_Conversation (CMCANC)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
				X	X*			X

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

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

X* In OS/2, this call is supported by Communications Server.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

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

Usage Notes

1. From the perspective of the local program, the conversation is terminated immediately. However, CPI Communications may not be able to notify the remote program until a later time.
2. A program is most likely to use the Cancel_Conversation call when a conversation with an outstanding operation must be terminated immediately.
3. The remote program will be notified of the termination of the conversation with the CM_DEALLOCATED_ABEND or CM_RESOURCE_FAILURE_RETRY return code or, if the conversation has *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and backout is required, with the CM_DEALLOCATED_ABEND_BO or CM_RESOURCE_FAILURE_RETRY_BO return code.

   Note: For half-duplex conversations, if the conversation is using an LU 6.2 CRM and the remote program has issued Send_Error with its end of the conversation in **Receive** state, the incoming information containing notice of CM_DEALLOCATED_ABEND is purged, and a CM_DEALLOCATED_NORMAL or CM_DEALLOCATED_NORMAL_BO return code is reported instead of CM_DEALLOCATED_ABEND or CM_DEALLOCATED_ABEND_BO, respectively. See “Send_Error (CMSERR)” on page 259 for a complete discussion.
4. Program-supplied log data is not sent to the remote system as a result of a Cancel_Conversation call.
5. When Cancel_Conversation is issued for a protected conversation, the conversation's context may be placed in the **Backout-Required** condition.
6. If the Cancel_Conversation call is the first operation on the conversation following an Accept (CMACCP) or Accept_Incoming (CMACCI) call and an OSI TP CRM is being used, then any initialization data specified by the use of the Set_Initialization_Data (CMSID) or Set_Mapped_Initialization_Data (CMSMID) call is sent to the remote program.

Related Information

“Non-Blocking Operations” on page 47 discusses the use of non-blocking operations.

“Extract_Mapped_Initialization_Data (CMEMID)” on page 170 discusses the extract initialization_data conversation characteristic for mapped initialization data.

“Set_Initialization_Data (CMSID)” on page 312 and “Extract_Initialization_Data (CMEID)” on page 168 discuss the *initialization_data* conversation characteristic.

“Set_Mapped_Initialization_Data (CMSMID)” on page 318 discusses the initialization_data conversation characteristic for mapped initialization data.

“Wait_For_Conversation (CMWAIT)” on page 369 and “Wait_For_Completion (CMWCMP)” on page 366 describe the normal completion of non-blocking operations.

Confirm (CMCFM)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

The Confirm (CMCFM) call is used by a local program to send a confirmation request to the remote program and then wait for a reply. The remote program replies with a Confirmed (CMCFMD) call. The local and remote programs use the Confirm and Confirmed calls to synchronize their processing of data.

Notes:

1. The *sync_level* conversation characteristic for the *conversation_ID* specified must be set to CM_CONFIRM or CM_SYNC_POINT to use this call. The Set_Sync_Level (CMSSL) call is used to set a conversation's synchronization level.
2. The Confirm call can be issued only on a half-duplex conversation.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

control_information_received *(output)*
Specifies the variable containing an indication of whether or not control information has been received.

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

- CM_NO_CONTROL_INFO_RECEIVED
  Indicates that no control information was received.
- CM_REQ_TO_SEND_RECEIVED (half-duplex conversations only)
  The local program received a request-to-send notification from the remote program. The remote program issued Request_To_Send, requesting the local program's end of the conversation to enter **Receive** state, which would place the remote program's end of the conversation in **Send** state. See "Request_To_Send (CMRTS)" on page 246 for further discussion of the local program's possible responses.
- CM_ALLOCATE_CONFIRMED (OSI TP CRM only)
  The local program received confirmation of the remote program's acceptance of the conversation.

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

  This value will be returned with a return code of CM_OK. The program will receive a CM_DEALLOCATED_ABEND return code on a later call on the conversation.
- CM_EXPEDITED_DATA_AVAILABLE (LU 6.2 CRM only)
  Expedited data is available to be received.
- CM_RTS_RCVD_AND_EXP_DATA_AVAIL (half-duplex conversations and LU 6.2 CRM only)
  The local program received a request-to-send notification from the remote program and expedited data is available to be received.

Notes:

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

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

- CM_OK (remote program replied Confirmed)
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_CONVERSATION_TYPE_MISMATCH
- CM_PIP_NOT_SPECIFIED_CORRECTLY
- CM_SECURITY_NOT_VALID
- CM_SYNC_LVL_NOT_SUPPORTED_PGM
- CM_SYNC_LVL_NOT_SUPPORTED_SYS
- CM_SEND_RCV_MODE_NOT_SUPPORTED
- CM_TPN_NOT_RECOGNIZED
- CM_TP_NOT_AVAILABLE_NO_RETRY
- CM_TP_NOT_AVAILABLE_RETRY
- CM_DEALLOCATED_ABEND
- CM_PROGRAM_ERROR_PURGING
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_DEALLOCATED_ABEND_SVC (basic conversations only)
- CM_DEALLOCATED_ABEND_TIMER (basic conversations only)

- CM_SVC_ERROR_PURGING (basic conversations only)
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  – The conversation is not in **Send**, **Send-Pending**, or **Defer-Receive** state.
  – The conversation is basic and in **Send** state, and the program started but did not finish sending a logical record.
  – For a conversation with *sync_level* set to CM_SYNC_POINT, the conversation's context is in the **Backout-Required** condition. The Confirm call is not allowed for this conversation while its context is in this condition.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  – The *sync_level* conversation characteristic is set to CM_NONE or CM_SYNC_POINT_NO_CONFIRM.
  – The *conversation_ID* specifies an unassigned conversation identifier.
  – The *send_receive_mode* of the conversation is CM_FULL_DUPLEX.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR
- The following values are returned only when *sync_level* is set to CM_SYNC_POINT:
  – CM_TAKE_BACKOUT
  – CM_DEALLOCATED_ABEND_BO
  – CM_DEALLOCATED_ABEND_SVC_BO (basic conversations only)
  – CM_DEALLOCATED_ABEND_TIMER_BO (basic conversations only)
  – CM_RESOURCE_FAIL_NO_RETRY_BO
  – CM_RESOURCE_FAILURE_RETRY_BO
  – CM_INCLUDE_PARTNER_REJECT_BO

State Changes

When *return_code* is set to CM_OK:

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

Usage Notes

1. The program that issues Confirm waits until a reply from the remote partner program is received. (This reply is made using the Confirmed call.)
2. The program can use this call for various application-level functions. For example:
   - The program can issue this call immediately following an Allocate call to determine if the conversation was allocated before sending any data.
   - The program can issue this call to determine if the remote program received the data sent. The remote program can respond by issuing a Confirmed call if it received and processed the data without error, or by issuing a Send_Error call if it encountered an error. The only other valid response from the remote program is the issuance of the Deallocate call

with *deallocate_type* set to CM_DEALLOCATE_ABEND or the Cancel_Conversation call.

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

Related Information

"Example 5: Validation of Data Receipt" on page 78 shows an example program using the Confirm call.

"Confirmed (CMCFMD)" on page 137 provides information on the remote program's reply to the Confirm call.

"Request_To_Send (CMRTS)" on page 246 provides a complete discussion of the *control_information_received* parameter.

"Set_Allocate_Confirm (CMSAC)" on page 282 describes how a program can request that the remote program confirm its acceptance of the conversation.

"Set_Sync_Level (CMSSL)" on page 354 explains how programs specify the level of synchronization processing.

Confirmed (CMCFMD)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

A program uses the Confirmed (CMCFMD) call to send a confirmation reply to the remote program. The local and remote programs can use the Confirmed and Confirm calls to synchronize their processing.

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

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

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

- CM_OK
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - For a half-duplex conversation, the conversation is not in **Confirm**, **Confirm-Send**, or **Confirm-Deallocate** state.
  - For a full-duplex conversation, the conversation is not in **Confirm-Deallocate** state.
  - For a conversation with *sync_level* set to CM_SYNC_POINT, the conversation's context is in the **Backout-Required** condition. The Confirmed call is not allowed for this conversation while its context is in this condition.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *send_receive_mode* is set to CM_FULL_DUPLEX and the conversation is using an LU 6.2 CRM.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

For a half-duplex conversation, when *return_code* is set to CM_OK:

- The conversation enters **Receive** state if the program received the *status_received* parameter set to CM_CONFIRM_RECEIVED on the preceding Receive call—that is, if the conversation was in **Confirm** state.
- The conversation enters **Send** state if the program received the *status_received* parameter set to CM_CONFIRM_SEND_RECEIVED on the preceding Receive call — that is, if the conversation was in **Confirm-Send** state.
- The conversation enters **Reset** state if the program received the *status_received* parameter set to CM_CONFIRM_DEALLOC_RECEIVED on the preceding Receive call—that is, if the conversation was in **Confirm-Deallocate** state.

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

Usage Notes

1. For a half-duplex conversation, the local program can issue this call only as a reply to a confirmation request; the call cannot be issued at any other time. A confirmation request is generated (by the remote system) when the remote program makes a call to Confirm. The remote program that has issued Confirm will wait until the local program responds with Confirmed.
2. For a half-duplex conversation, the program can use this call for various application-level functions. For example, the remote program may send data followed by a confirmation request (using the Confirm call). When the local program receives the confirmation request, it can issue a Confirmed call to indicate that it received and processed the data without error.

Related Information

"Example 5: Validation of Data Receipt" on page 78 shows an example program using the Confirmed call.

"Confirm (CMCFM)" on page 133 provides more information on the Confirm call.

"Receive (CMRCV)" on page 213 provides more information on the *status_received* parameter.

"Set_Sync_Level (CMSSL)" on page 354 explains how programs specify the level of synchronization processing.

Convert_Incoming (CMCNVI)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X*				X	X*	X		X

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

X* In AIX, this call is supported in Version 3 Release 1 or later. In OS/2, this call is supported by Communications Server.

Format

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

Parameters

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

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

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *buffer_length* is invalid for the range permitted by the implementation.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call causes no state changes.

Usage Notes

1. When the EBCDIC hexadecimal codes, specified in Table 60 on page 647, represent the encoding for the data transmitted across the network, the Convert_Incoming call can be used to convert the EBCDIC hexadecimal codes to the corresponding local representation of the data.

2. Convert_Incoming converts data on a character-by-character basis. Since the program may use character values beyond those defined in Table 60 on page 647, care must be taken in the use of Convert_Incoming in that it may

generate implementation-dependent results if applied to a string that contains such values.

Networking Services for Windows converts the values to X'FF'.

3. A program may be written to be independent of the encoding (such as ASCII or EBCDIC) of the partner program by sending and receiving EBCDIC data records with the help of the Convert_Outgoing and Convert_Incoming calls. The sending program calls Convert_Outgoing to convert the data record to EBCDIC before sending it. The receiving program calls Convert_Incoming to convert the EBCDIC data record to the appropriate encoding for its environment.
4. If the local encoding is EBCDIC, the Convert_Incoming call does not change the buffer contents.

Related Information

"Data Conversion" on page 43 provides information about data conversion and the Convert_Incoming call.

"Convert_Outgoing (CMCNVO)" on page 141 provides information about the Convert_Outgoing call.

"Receive_Mapped_Data (CMRCVM)" on page 231 provides information about receiving mapped partner data.

"Send_Mapped_Data (CMSNDM)" on page 271 provides information about sending mapped partner data.

Convert_Outgoing (CMCNVO)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X*				X	X*	X		X

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

X* In AIX, this call is supported in Version 3 Release 1 or later. In OS/2, this call is supported by Communications Server.

Format

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

Parameters

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

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

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *buffer_length* is invalid for the range permitted by the implementation.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call causes no state changes.

Usage Notes

1. When the EBCDIC hexadecimal codes, specified in Table 60 on page 647, represent the encoding for the data transmitted across the network, the Convert_Outgoing call can be used to convert the data supplied by the program from the local encoding to the corresponding EBCDIC hexadecimal codes.

2. Convert_Outgoing converts data on a character-by-character basis. Since the program may use character values beyond those defined in Table 60 on page 647, care must be taken in the use of Convert_Outgoing in that it may

generate implementation-dependent results if applied to a string which contains such values.

Networking Services for Windows converts the values to X'FF'.

3. A program may be written to be independent of the encoding (such as ASCII or EBCDIC) of the partner program by sending and receiving EBCDIC data records with the help of the Convert_Outgoing and Convert_Incoming calls. The sending program calls Convert_Outgoing to convert the data record to EBCDIC before sending it. The receiving program calls Convert_Incoming to convert the EBCDIC data record to the appropriate encoding for its environment.
4. If the local encoding is EBCDIC, the Convert_Outgoing call does not change the buffer contents.

Related Information

"Data Conversion" on page 43 provides information about data conversion and the Convert_Outgoing call.

"Convert_Incoming (CMCNVI)" on page 139 provides information about the Convert_Incoming call.

"Receive_Mapped_Data (CMRCVM)" on page 231 provides information about receiving mapped partner data.

"Send_Mapped_Data (CMSNDM)" on page 271 provides information about sending mapped partner data.

Deallocate (CMDEAL)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

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

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

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

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

CALL CMSDT – Set_Deallocate_Type
CALL CMSLD – Set_Log_Data

Format

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

Parameters

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

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

The following return codes apply to half-duplex conversations.

For any of the following conditions:

– *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and either *sync_level* is set to CM_NONE or the conversation is in **Initialize-Incoming** state
– *deallocate_type* is set to CM_DEALLOCATE_FLUSH
– *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, but the conversation is not currently included in a transaction

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

- CM_OK (deallocation is completed)
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  – The conversation is not in **Send**, **Send-Pending** or **Initialize-Incoming** state.
  – The conversation is basic and in **Send** state; and the program started but did not finish sending a logical record.
  – The *deallocate_type* is set to CM_DEALLOCATE_FLUSH, and the conversation is currently included in a transaction.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

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

- CM_OK (deallocation is completed)
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_OPERATION_NOT_ACCEPTED
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_PRODUCT_SPECIFIC_ERROR

For any of the following conditions:

– *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and the *sync_level* is set to CM_CONFIRM
– *deallocate_type* is set to CM_DEALLOCATE_CONFIRM
– *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT, but the conversation is not currently included in a transaction

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

- CM_OK (deallocation is completed)
- CM_OPERATION_INCOMPLETE

- CM_CONVERSATION_CANCELLED
- CM_CONVERSATION_TYPE_MISMATCH
- CM_PIP_NOT_SPECIFIED_CORRECTLY
- CM_SECURITY_NOT_VALID
- CM_SYNC_LVL_NOT_SUPPORTED_PGM
- CM_SYNC_LVL_NOT_SUPPORTED_SYS
- CM_SEND_RCV_MODE_NOT_SUPPORTED
- CM_TPN_NOT_RECOGNIZED
- CM_TP_NOT_AVAILABLE_NO_RETRY
- CM_TP_NOT_AVAILABLE_RETRY
- CM_DEALLOCATED_ABEND
- CM_PROGRAM_ERROR_PURGING
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_DEALLOCATED_ABEND_SVC (basic conversations only)
- CM_DEALLOCATED_ABEND_TIMER (basic conversations only)
- CM_SVC_ERROR_PURGING
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The conversation is not in **Send** or **Send-Pending** state.
  - The conversation is basic and in **Send** state; and the program started but did not finish sending a logical record.
  - The *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, and the conversation is currently included in a transaction.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

If the *deallocate_type* conversation characteristic is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and the conversation is included in a transaction, the *return_code* variable can have one of the following values:

- CM_OK
  Deallocation is deferred until the program issues a resource recovery commit call. If the commit call is successful, the conversation will be deallocated normally. If the commit is not successful or if the program issues a resource recovery backout call instead of a commit, the conversation will not be deallocated. Instead, the conversation will be restored to the state it was in at the previous synchronization point.

  Table 70 on page 710 and Table 71 on page 711 show how resource recovery calls affect CPI Communications conversation states.
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The conversation is not in **Send** or **Send-Pending** state.
  - The conversation is basic and in **Send** state, and the program started but did not finish sending a logical record.
  - The conversation's context is in the **Backout-Required** condition.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:

  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The conversation is using an OSI TP CRM, and the program is not the superior for the conversation.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

The following return codes apply to full-duplex conversations.

For any of the following conditions:

- *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and either *sync_level* is set to CM_NONE or the conversation is in **Initialize-Incoming** state
- *deallocate_type* is set to CM_DEALLOCATE_FLUSH
- *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, but the conversation is not currently included in a transaction

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

- CM_OK
  Deallocation is completed if this call was issued in **Send-Only** or **Initialize-Incoming** state. Otherwise, the call was issued in **Send-Receive** state and the conversation goes to **Receive-Only** state. In this latter case, the conversation will later get deallocated when a terminating error condition occurs, either this program or the partner program deallocates the conversation abnormally or cancels it, or the partner program deallocates the conversation using the function of the Flush call.
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The conversation is not in **Send-Receive** or **Send-Only** or **Initialize-Incoming** state.
  - The conversation is basic and in **Send-Receive** or **Send-Only** state, and the program started but did not finish sending a logical record.
  - The local program has received a *status_received* value of CM_JOIN_TRANSACTION and must issue a tx_begin call to the X/Open TX interface to join the transaction.
  - The *deallocate_type* is set to CM_DEALLOCATE_FLUSH, and the conversation is currently included in a transaction.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_ALLOCATION_ERROR
- CM_DEALLOCATED_ABEND
- CM_DEALLOCATED_ABEND_SVC
- CM_DEALLOCATED_ABEND_TIMER
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_DEALLOCATED_NORMAL
- CM_OPERATION_NOT_ACCEPTED
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_PRODUCT_SPECIFIC_ERROR

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

- CM_OK (deallocation is completed)
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_OPERATION_NOT_ACCEPTED
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_PRODUCT_SPECIFIC_ERROR

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

- CM_OK (deallocation is completed)
- CM_DEALLOCATE_CONFIRM_REJECT
  This value indicates that the partner program rejected the confirmation request. The conversation is not deallocated.
- CM_ALLOCATION_ERROR
- CM_DEALLOCATED_ABEND
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_DEALLOCATED_NORMAL
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The conversation is not in **Send-Receive** state.
  - The conversation is basic and in **Send-Receive** state, and the program started but did not finish sending a logical record.
  - The local program has received a *status_received* value of CM_JOIN_TRANSACTION and must issue a tx_begin call to the X/Open TX interface to join the transaction.
  - The *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, and the conversation is currently included in a transaction.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PRODUCT_SPECIFIC_ERROR
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_OPERATION_NOT_ACCEPTED

If the *deallocate_type* conversation characteristic is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, and the conversation is included in a transaction, the *return_code* variable can have one of the following values:

- CM_OK
  Deallocation is deferred until the program issues a resource recovery commit call. If the commit call is successful, the conversation will be deallocated normally. If the commit is not successful or if the program issues a resource recovery backout call instead of a commit, the conversation will not be deallocated. Instead, the conversation will be restored to the state it was in at the previous synchronization point.

Table 70 on page 710 and Table 71 on page 711 show how resource recovery calls affect CPI Communications conversation states.

- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The conversation is not in **Send-Receive** state.
  - The conversation is basic and in **Send-Receive** state, and the program started but did not finish sending a logical record.
  - The conversation's context is in the **Backout-Required** condition.
  - The local program has received a *status_received* value of CM_JOIN_TRANSACTION and must issue a tx_begin call to the X/Open TX interface to join the transaction.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The conversation is using an OSI TP CRM, and the program is not the superior for the conversation.
- CM_ALLOCATION_ERROR
- CM_TAKE_BACKOUT
- CM_DEALLOCATED_ABEND_BO
- CM_DEALLOCATED_ABEND_SVC_BO
- CM_DEALLOCATED_ABEND_TIMER_BO
- CM_RESOURCE_FAILURE_RETRY_BO
- CM_RESOURCE_FAIL_NO_RETRY_BO
- CM_INCLUDE_PARTNER_REJECT_BO
- CM_CONV_DEALLOC_AFTER_SYNCPT
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

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

- The conversation enters **Reset** state if *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and either *sync_level* is set to CM_NONE or the conversation is in **Initialize-Incoming** state, or if *deallocate_type* is set to CM_DEALLOCATE_FLUSH, CM_DEALLOCATE_CONFIRM, or CM_DEALLOCATE_ABEND.

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

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

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

- The conversation enters **Reset** state if *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and either *sync_level* is set to CM_NONE or the conversation is in **Initialize-Incoming** state, or if *deallocate_type* is set to CM_DEALLOCATE_CONFIRM or CM_DEALLOCATE_ABEND.

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

Usage Notes

1. The execution of Deallocate includes the flushing of the local system's send buffer if any of the following conditions is true:
   - *deallocate_type* is set to CM_DEALLOCATE_FLUSH or CM_DEALLOCATE_CONFIRM
   - *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is CM_NONE or CM_CONFIRM
   - *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, but the conversation is not currently included in a transaction.

   If *deallocate_type* is CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and the conversation is included in a transaction, the local system's send buffer will not be flushed until a resource recovery commit or backout call is issued by the program or the sync point manager.
2. If a conversation has *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, CPI Communications does not allow the conversation to be deallocated with a *deallocate_type* of CM_DEALLOCATE_CONFIRM or CM_DEALLOCATE_FLUSH unless *transaction_control* is set to CM_UNCHAINED_TRANSACTIONS and the conversation is not currently included in a transaction.
3. If *deallocate_type* is set to CM_DEALLOCATE_ABEND and the *log_data_length* characteristic is greater than zero, the system formats the supplied log data into the appropriate format. The data supplied by the program is any data the program wants to have logged. The data is logged on the local system's error log and is also sent to the remote system for logging there.
4. The remote program receives the deallocate notification by means of a *return_code* or *status_received* indication, as follows:
   - CM_DEALLOCATED_NORMAL *return_code*
     This return code indicates that the local program issued Deallocate with the *deallocate_type* set to CM_DEALLOCATE_FLUSH, or with the *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* set to CM_NONE, or with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* set to

CM_SYNC_POINT_NO_CONFIRM, and the conversation not currently included in a transaction.

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

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

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

  Note: For a half-duplex conversation, if the conversation is using an LU 6.2 CRM and the remote program has issued Send_Error with its end of the conversation in **Receive** state, the incoming information containing notice of CM_DEALLOCATED_ABEND is purged and a CM_DEALLOCATED_NORMAL *return_code* is reported instead of CM_DEALLOCATED_ABEND. See "Send_Error (CMSERR)" on page 259 for a complete discussion.

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

  Note: For a half-duplex conversation, if the conversation is using an LU 6.2 CRM and the remote program has issued Send_Error with its end of the conversation in **Receive** state, the incoming information containing notice of CM_DEALLOCATED_ABEND_BO is purged and a CM_DEALLOCATED_NORMAL_BO *return_code* is reported instead of CM_DEALLOCATED_ABEND_BO. See "Send_Error (CMSERR)" on page 259 for a complete discussion.

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

- CM_TAKE_COMMIT_DEALLOCATE *status_received* indication
  This indicates that the local program issued a resource recovery commit call after issuing a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and with the conversation included in a transaction.

5. The program should terminate all conversations before the end of the program. However, if the program does not terminate all conversations, node services will abnormally deallocate any dangling conversations. The way abnormal deallocation is accomplished is implementation-specific.

6. When a Deallocate call is issued with *deallocate_type* set to CM_DEALLOCATE_ABEND and *sync_level* set to CM_SYNC_POINT or

CM_SYNC_POINT_NO_CONFIRM, the conversation's context may be placed in the **Backout-Required** condition.

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

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

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

   At the local program, all outstanding operations are terminated. No guarantee is given on the results of the terminated operation.

   At the remote program,

   - Other than Confirmed and Set calls, a new or outstanding call associated with the Send queue gets CM_DEALLOCATED_ABEND, and the conversation goes to **Receive-Only** or **Reset** state if it was in **Send-Receive** or **Send-Only** state, respectively.
   - Any data sent by the partner before it issued the Deallocate call can be received, after which the next Receive call will get CM_DEALLOCATED_ABEND. The conversation is now in **Reset** state.
   - Calls associated with the Expedited-Send queue until the conversation goes to **Reset** state get CM_CONVERSATION_ENDING. An outstanding call is terminated when the conversation goes to **Reset** state. No guarantee is given on the results of the terminated operation.
   - Calls associated with the Expedited-Receive queue until the conversation goes to **Reset** state get CM_CONVERSATION_ENDING after any available expedited data has been received. An outstanding call is terminated when the conversation goes to **Reset** state. No guarantee is given on the results of the terminated operation.

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

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

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

    – *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is set to CM_NONE
    – *deallocate_type* is set to CM_DEALLOCATE_FLUSH

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

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

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

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

Related Information

"Example 1: Data Flow in One Direction" on page 69 shows an example program flow using the Deallocate call for a half-duplex conversation.

"Example 10: Terminating a Full-Duplex Conversation" on page 88 shows how a full-duplex conversation can be deallocated.

"Set_Deallocate_Type (CMSDT)" on page 303 discusses the *deallocate_type* characteristic and its possible values.

"Set_Log_Data (CMSLD)" on page 316 discusses the *log_data* characteristic.

Deferred_Deallocate (CMDFDE)

OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32

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

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

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

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - This value indicates the *conversation_ID* specifies an unassigned identifier.
  - The conversation is not using an OSI TP CRM.
  - The program is not the superior for the conversation.
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The conversation is not in **Send** or **Send-Pending** state (for half-duplex conversations) or **Send-Receive** state (for full-duplex conversations).
  - The conversation is not currently included in a transaction.
- CM_TAKE_BACKOUT
- CM_DEALLOCATED_ABEND_BO
- CM_RESOURCE_FAILURE_RETRY_BO
- CM_RESOURCE_FAIL_NO_RETRY_BO
- CM_INCLUDE_PARTNER_REJECT_BO
- CM_OPERATION_NOT_ACCEPTED

- CM_PRODUCT_SPECIFIC_ERROR

The following return codes apply to half-duplex conversations.

- CM_SYNC_LVL_NOT_SUPPORTED_SYS
- CM_SEND_RCV_MODE_NOT_SUPPORTED
- CM_TPN_NOT_RECOGNIZED
- CM_TP_NOT_AVAILABLE_NO_RETRY
- CM_TP_NOT_AVAILABLE_RETRY
- CM_PROGRAM_ERROR_PURGING

The following return code applies to full-duplex conversations.

- CM_ALLOCATION_ERROR

State Changes

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

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

Usage Notes

1. Once a commit call completes successfully for this transaction, a Deferred_Deallocate call performs the same function as a Deallocate (CMDEAL) call with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* set to either CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.
2. If the transaction does not complete successfully and is backed out, longer in effect.

Related Information

"Deallocate (CMDEAL)" on page 143 discusses conversation deallocation in more detail.

Extract_AE_Qualifier (CMEAEQ)

OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32

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

The *AE_qualifier* conversation characteristic identifies the OSI application-entity qualifier associated with the remote program.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

AE_qualifier *(output)*
Specifies the variable containing the application-entity-qualifier for the remote program. The variable must be defined with a length of at least 1024 bytes.

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

AE_qualifier_length *(output)*
Specifies the variable containing the length of the returned *AE_qualifier* parameter.

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

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

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

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

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned identifier.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the *conversation_ID* specifies a conversation in **Initialize-Incoming** state.
- CM_OPERATION_NOT_ACCEPTED
  This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

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

Related Information

"Set_AE_Qualifier (CMSAEQ)" on page 280 and "Side Information" on page 23 provide more information on the *AE_qualifier* conversation characteristic.

Extract_AP_Title (CMEAPT)

OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32

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

The *AP_title* conversation characteristic identifies the OSI application-process title associated with the remote program.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

AP_title *(output)*
Specifies the variable containing the title of the application-process where the remote program is located. The variable must be defined with a length of at least 1024 bytes.

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

AP_title_length *(output)*
Specifies the variable containing the length of the returned *AP_title* parameter.

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

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

- CM_DN
  Specifies that the *AP_title* is a distinguished name.
- CM_OID
  Specifies that the *AP_title* is an object identifier.

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

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned identifier.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the *conversation_ID* specifies a conversation in **Initialize-Incoming** state.
- CM_OPERATION_NOT_ACCEPTED
  This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

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

Related Information

"Set_AP_Title (CMSAPT)" on page 284 and "Side Information" on page 23 provide more information on the *AP_title* conversation characteristic.

Extract_Application_Context_Name (CMEACN)

OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32

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

The *application_context_name* conversation characteristic identifies the OSI application context used on the conversation.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

application_context_name *(output)*
Specifies the application context name that is to be used on the conversation. The variable must be defined with a length of at least 256 bytes.

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

application_context_name_length *(output)*
Specifies the length of the application context name that is to be used on the conversation.

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

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned identifier.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the *conversation_ID* specifies a conversation in **Initialize-Incoming** state.

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

State Changes

This call does not cause any state changes.

Usage Notes

1. This call does not change the *application_context_name* or the *application_context_name_length* conversation characteristic for the specified conversation.
2. The application context name is an object identifier and is represented as a series of digits separated by periods. For example, the application context defined by the Open Systems Environment Implementers' Workshop (OIW) for UDT with Commit Profiles is represented as "1.3.14.15.5.1.0" and the application context for Application Supported Transactions using UDT is represented as "1.3.14.15.5.2.0".
3. The call is not associated with any conversation queue. When a conversation uses queue-level non-blocking, the call does not return CM_OPERATION_NOT_ACCEPTED on the conversation.

Related Information

"Set_Application_Context_Name (CMSACN)" on page 286 and "Side Information" on page 23 provide more information on the *application_context_name* conversation characteristic.

Extract_Conversation_Context (CMECTX)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X*					X*			X

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

The *conversation_context* conversation characteristic identifies which partner program is associated with the work being done by the local program.

X* In AIX, this call is supported in Version 3 Release 1 or later. In OS/2, this call is supported in Communications Server.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

context_ID *(output)*
Specifies the variable containing the context identifier for this conversation. The variable must be defined with a length of at least 32 bytes.

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

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

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

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is in **Initialize** or **Initialize-Incoming** state.

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

State Changes

This call does not cause a state change.

Usage Notes

1. This call does not change the *context_ID* for the specified conversation.
2. A program may want to determine the context for a conversation prior to allocating a new conversation. The program would issue Extract_Conversation_Context to get the *context_ID* and then would use node services to set that context as the current context prior to issuing the Allocate call for the new conversation.
3. The call is not associated with any conversation queue. When a conversation uses queue-level non-blocking, the call does not return CM_OPERATION_NOT_ACCEPTED on the conversation.

Related Information

"Contexts and Context Management" on page 32 defines contexts and discusses how application programs can manage multiple contexts.

Extract_Conversation_State (CMECS)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

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

The *conversation_state* conversation characteristic identifies the state of the conversation with respect to the local program.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

conversation_state *(output)*
Specifies the conversation state that is returned to the local program. For half-duplex conversations, the *conversation_state* can be one of the following:

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

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

- CM_INITIALIZE_STATE
- CM_CONFIRM_DEALLOCATE_STATE
- CM_DEFER_DEALLOCATE_STATE
- CM_SYNC_POINT_STATE
- CM_SYNC_POINT_DEALLOCATE_STATE

- CM_INITIALIZE_INCOMING_STATE
- CM_SEND_ONLY_STATE
- CM_RECEIVE_ONLY_STATE
- CM_SEND_RECEIVE_STATE
- CM_PREPARED_STATE

Notes:

- Unless *return_code* is set to CM_OK, the value of *conversation_state* is not meaningful.
- The following *conversation_state* values are returned only on CICS, OS/400, and VM, because these systems support a *sync_level* of CM_SYNC_POINT:
  - CM_DEFER_RECEIVE_STATE
  - CM_DEFER_DEALLOCATE_STATE
  - CM_SYNC_POINT_STATE
  - CM_SYNC_POINT_SEND_STATE
  - CM_SYNC_POINT_DEALLOCATE_STATE

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

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
  This return code indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_TAKE_BACKOUT
  This value is returned only when all of the following conditions are true:
  - The *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and the conversation is included in a transaction.
  - The conversation is not in **Initialize**, **Initialize-Incoming**, **Confirm-Deallocate**, **Send-Only**, or **Receive-Only** state.
  - The conversation's context is in the **Backout-Required** condition.
  - The program is using protected resources that must be backed out.
- CM_OPERATION_NOT_ACCEPTED
  This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

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

Related Information

"Support for Resource Recovery Interfaces" on page 54 provides more information on using resource recovery interfaces.

Extract_Conversation_Type (CMECT)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

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

The *conversation_type* conversation characteristic identifies whether the data exchange on the conversation will follow basic or mapped rules.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

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

- CM_BASIC_CONVERSATION
  Indicates that the conversation is allocated as a basic conversation.
- CM_MAPPED_CONVERSATION
  Indicates that the conversation is allocated as a mapped conversation.

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

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

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
  This return code indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the *conversation_ID* specifies a conversation in **Initialize-Incoming** state.
- CM_OPERATION_NOT_ACCEPTED
  This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

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

Related Information

"Set_Conversation_Type (CMSCT)" on page 301 provides more information on the *conversation_type* characteristic.

Extract_Initialization_Data (CMEID)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32

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

The *initialization_data* conversation characteristic contains program-supplied data that is sent to, or received from, the remote program during conversation initialization.

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

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

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

initialization_data *(output)*
Specifies the variable containing the initialization data. Initialization data may be from 0 to 10000 bytes.

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

requested_length *(input)*
Specifies the length of the *initialization_data* variable to contain the initialization data.

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

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  – The *conversation_ID* specifies an unassigned identifier.
  – The *requested_length* specifies a value less than 0.
- CM_BUFFER_TOO_SMALL
  The *requested_length* specifies a value that is less than the size of the *initialization_data* characteristic to be extracted. The *initialization_data* characteristic is not returned, but the *initialization_data_length* parameter is set to indicate the size required.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the *conversation_ID* specifies a conversation in **Initialize-Incoming** state.
- CM_OPERATION_NOT_ACCEPTED
  This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

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

Related Information

“Extract_Mapped_Initialization_Data (CMEMID)” on page 170 describes information about extracting the initialization_data conversation characteristic for mapped initialization data.

“Set_Mapped_Initialization_Data (CMSMID)” on page 318 describes how the initialization data is set by the remote program.

Extract_Mapped_Initialization_Data (CMEMID)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32

A program uses the Extract_Mapped_Initialization_Data (CMEMID) call to extract the *initialization_data* and *initialization_data_length* conversation characteristics received from the remote program for a given conversation after they have been processed by the underlying map routine. The *map_name* identifies the data so that it can be processed properly. The values are returned to the program in the *initialization_data* and *initialization_data_length* parameters.

The *initialization_data* conversation characteristic contains program-supplied data that is sent to, or received from, the remote program during conversation initialization.

The Extract_Mapped_Initialization_Data call is used by the recipient of the conversation to extract the incoming initialization data received from the initiator of the conversation and transformed by the underlying map routine. It may be issued following the Accept_Conversation or Accept_Incoming call.

The Extract_Mapped_Initialization_Data call may be used by the initiator of the conversation to extract the incoming initialization data from the recipient of the conversation after it has been processed by the U-ASE. The Extract_Initialization_Data call may also be used by the initiator of the conversation to extract the incoming initialization data from the recipient of the conversation. In this case, the call may be issued following receipt of a *control_information_received* value of CM_ALLOCATE_CONFIRMED_WITH_DATA or CM_ALLOCATE_REJECTED_WITH_DATA.

Format

```
CALL CMEMID(conversation_ID,
            map_name,
            map_name_length,
            initialization_data,
            requested_length,
            initialization_data_length,
            return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

map_name *(output)*
Specifies the variable containing the mapping function used to decode the data record. The length of the variable must be at least 64 bytes.

map_name_length *(output)*
Specifies the variable containing the length of the returned_map name parameter.

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

requested_length *(input)*
Specifies the length of the *initialization_data* variable to contain the initialization data.

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

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned identifier.
  - The *requested_length* specifies a value less than 0.
  - The *conversation_type* characteristic is set to CM_BASIC_CONVERSATION.
- CM_BUFFER_TOO_SMALL
  The *requested_length* specifies a value that is less than the size of the *initialization_data* characteristic to be extracted. The *initialization_data* characteristic is not returned, but the *initialization_data_length* parameter is set to indicate the size required.
- CM_UNKNOWN_MAP_NAME_RECEIVED
  The received data requires a map routine that the local map function does not support. The buffer contains the unprocessed user data.
- CM_MAP_ROUTINE_ERROR
  The map routine encountered a problem with the received data. The buffer contains the unprocessed user data.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the *conversation_ID* specifies a conversation in **Initialize-Incoming** state.
- CM_OPERATION_NOT_ACCEPTED
  This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

1. This call does not change the value of the *initialization_data* or the *initialization_data_length* characteristic for the specified conversation.
2. The program that initiates the conversation (issues Initialize_Conversation) must set *allocate_confirm* to CM_ALLOCATE_CONFIRM if it is expecting initialization data to be returned from the remote program following its confirmation of acceptance of the conversation.
3. The call is not associated with any conversation queue. When a conversation uses queue-level non-blocking, the call does not return CM_OPERATION_NOT_ACCEPTED on the conversation.
4. The *received_length* is local information from the map routine. The receive_length value is calculated by the map routine after all transformations are completed.
5. If CM_BUFFER_TOO_SMALL is returned, *Initialization_data* wil have been set to the length required to receive the data. You can allocate a larger buffer in which to receive the data and reissue the call or perform other user specified actions.

Related Information

"Extract_Initialization_Data (CMEID)" on page 168 describes how the mapped initialization data is set by the remote program.

"Set_Initialization_Data (CMSID)" on page 312 describes how the set initialization data is sent to the remote program.

Extract_Maximum_Buffer_Size (CMEMBS)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X*				X	X*	X		X

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

The *maximum_buffer_size* conversation characteristic contains the value of the largest supported buffer size on the local system.

X* In AIX, this call is supported in Version 3 Release 1 or later. In Communications Manager/2, this call is supported in Version 1.11 or later.

Format

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

Parameters

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

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

- CM_OK
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call causes no state changes.

Usage Notes

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

The largest amount of data that can be received by the program in a Receive call is:

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

Extract_Mode_Name (CMEMN)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

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

The *mode_name* conversation characteristic is a name that is used to identify the network properties requested for the conversation. When the conversation is using an OSI TP CRM, the *mode_name* used during conversation initialization is not available to the recipient of the conversation.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

mode_name *(output)*
Specifies the variable containing the mode name. The mode name designates the network properties for the logical connection allocated, or to be allocated, which will carry the conversation specified by the *conversation_ID*.

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

mode_name_length *(output)*
Specifies the variable containing the length of the returned *mode_name* parameter. The variable must be defined with a length of at least 8 bytes.

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

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

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the *conversation_ID* specifies a conversation in **Initialize-Incoming** state.

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

State Changes

This call does not cause a state change.

Usage Notes

1. This call does not change the *mode_name* for the specified conversation.
2. CPI Communications returns the *mode_name* using the native encoding of the local system.
3. The call is not associated with any conversation queue. When a conversation uses queue-level non-blocking, the call does not return CM_OPERATION_NOT_ACCEPTED on the conversation.

Related Information

"Automatic Conversion of Characteristics" on page 41 provides further information on the automatic conversion of the *mode_name* parameter.

"Set_Mode_Name (CMSMN)" on page 321 and "Side Information" on page 23 provide further information on the *mode_name* characteristic.

Extract_Partner_ID (CMEPID)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32

Programs use the Extract_Partner_ID call to determine the *partner_ID* characteristic for a conversation.

Format

```
CALL CMEPID(conversation_ID,
            partner_ID_type,
            partner_ID,
            requested_length,
            partner_ID_length,
            partner_ID_scope,
            directory_syntax,
            directory_encoding,
            return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation ID.

partner_ID_type *(output)*
Specifies the type of the *partner_ID* characteristic. If the call is issued after Allocate completes, the *partner_ID_type* will be of type CM_PROGRAM_BINDING.

partner_ID *(output)*
Specifies the *partner_ID* characteristic for the conversation. This parameter is only set if the *return_code* is CM_OK.

requested_length *(input)*
The input value of this parameter specifies the maximum size of the *partner_ID* variable the program is to receive.

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

partner_ID_scope *(output)*
Specifies the scope of the search in the use of the *partner_ID*. The *partner_ID_scope* variable only has meaning when the *partner_ID* is of type CM_DISTINGUISHED_NAME.

directory_syntax *(output)*
Specifies the syntax of the distributed directory that CPI Communications accesses with the *partner_ID*. This parameter has no meaning if *partner_ID_type* is CM_PROGRAM_BINDING.

directory_encoding *(output)*
Specifies the encoding rule of the distributed directory that CPI Communications accesses with the *partner_ID*. This parameter has no meaning if *partner_ID_type* is CM_PROGRAM_BINDING.

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is in **Initialize-Incoming** state.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *requested_length* specifies a value less than 0.
- CM_BUFFER_TOO_SMALL
  The *requested_length* specifies a value that is less than the size of the *partner_ID* characteristic to be extracted. The *partner_ID* characteristic is not returned, but the *partner_ID_length* parameter is set to indicate the size required.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. Unless the *return_code* indicates CM_OK or CM_BUFFER_TOO_SMALL, the values of all other parameters on this call have no meaning.

2. If the value specified for the *requested_length* is less than the size of the *partner_ID* characteristic to be extracted, the program receives a return code of CM_BUFFER_TOO_SMALL and the *partner_ID* is not returned. The *partner_ID_length* parameter, however, is updated to show the size *partner_ID* parameter required. A subsequent call to Extract_Partner_ID with sufficient size completes successfully.

3. When used before the Allocate call, this call returns the *partner_ID* value established from side information or by a prior Set_Partner_ID call. After a successful Allocate call, Extract_Partner_ID returns the program binding used to establish the conversation. The program can determine the CRM type on which the conversation is allocated by parsing the program binding. See "Program Binding" on page 658 for details.

4. If the call is issued by the program that accepts the conversation, the program binding returned to the program may not be complete. For example, programs accepting a conversation with a CRM type of LU 6.2 are not able to access the TP name of the partner program. In these cases, Extract_Partner_ID returns with a *partner_ID_type* of CM_PROGRAM_BINDING and the *partner_ID* variable contains as much information on the partner program as is currently available.

5. This call may be issued in **Reset** state to retrieve the program binding. When the call completes successfully, CPI Communications no longer keeps the *partner_ID*. The program should issue this call immediately upon entering

Reset state, as the duration of the program binding and valid conversation identifier is implementation-specific.

Related Information

"Distributed Directory" on page 25 explains the terms and concepts required for use of a distributed directory.

"Example 14: Using the Distributed Directory to Find the Partner Program" on page 96 provides a sample scenario of a program using the Extract_Partner_ID call.

"Set_Partner_ID (CMSPID)" on page 323 provides details on how to set the *partner_ID* characteristic.

"Program Binding" on page 658 describes the format of a program binding.

Extract_Partner_LU_Name (CMEPLN)

LU 6.2

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

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

The *partner_lu_name* conversation characteristic contains the SNA LU name that identifies the location of the remote program.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

partner_LU_name *(output)*
Specifies the variable containing the name of the LU where the remote program is located. The variable must be defined with a length of at least 17 bytes.

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

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

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

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the *conversation_ID* specifies a conversation in **Initialize-Incoming** state.

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

State Changes

This call does not cause a state change.

Usage Notes

1. This call does not change the *partner_LU_name* for the specified conversation.
2. CPI Communications returns the *partner_LU_name* using the native encoding of the local system.
3. The call is not associated with any conversation queue. When a conversation uses queue-level non-blocking, the call does not return CM_OPERATION_NOT_ACCEPTED on the conversation.

Related Information

"Automatic Conversion of Characteristics" on page 41 provides further information on the automatic conversion of the *partner_LU_name* parameter.

"Set_Partner_LU_Name (CMSPLN)" on page 327 and "Side Information" on page 23 provide more information on the *partner_LU_name* characteristic.

Extract_Secondary_Information (CMESI)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
					X			

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

OS/2 provides partial support for secondary information.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

call_ID *(input)*
Specifies the call identifier (see Table 59 on page 642).

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

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

requested_length *(input)*
Specifies the maximum amount of secondary information the program is to receive. Valid *requested_length* values range from 1 to the maximum buffer size supported by the system. The maximum buffer size is at least 32767 bytes. See Usage Note 13 on page 225 of the Receive call for additional information about determining the maximum buffer size.

data_received *(output)*
Specifies whether or not the program received complete secondary information.

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

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

- CM_COMPLETE_DATA_RECEIVED
  This value indicates that complete secondary information is received.
- CM_INCOMPLETE_DATA_RECEIVED

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

received_length *(output)*
Specifies the variable containing the amount of secondary information the program received, up to the maximum.

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

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_NO_SECONDARY_INFORMATION
  This value indicates that no secondary information is available for the specified call on the specified conversation.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  – The *conversation_ID* specifies an unassigned conversation identifier.
  – The *call_ID* specifies CM_CMESI or an undefined value.
  – The *requested_length* specifies a value that exceeds the range permitted by the implementation.

State Changes

This call does not cause a state change.

Usage Notes

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

  - On a per-thread basis for non-conversation calls
- For conversations not using conversation-level non-blocking, secondary information is kept:
  - On a per-queue basis for calls that are associated with a conversation queue
  - On a per-thread basis for calls that are not associated with any conversation queue

Related Information

“Extract_Maximum_Buffer_Size (CMEMBS)” on page 173 further discusses determining the maximum buffer size supported by the system.

“Secondary Information” on page 679 provides a complete discussion of secondary information.

Extract_Security_User_ID (CMESUI)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X*				X	X*	X	X*	X

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

The *security_user_id* conversation characteristic contains the user identification associated with the conversation.

X* AIX, prior to Version 3 Release 1, supported this function in a product-specific extension call.
OS/2, prior to Communications Server, supported this function in a product-specific extension call.
VM currently supports this function in a product-specific extension call.
For AIX, see "Extract_Conversation_Security_User_ID (XCECSU)" on page 398.
For OS/2, see "Extract_Conversation_Security_User_ID (XCECSU)" on page 612.
For VM, see "Extract_Conversation_Security_User_ID (XCECSU)" on page 546.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

security_user_ID *(output)*
Specifies the variable containing the user ID. The variable must be defined with a length of at least 10 bytes.

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

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

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

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

- CM_OK

- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the program is in **Initialize-Incoming** state.
- CM_OPERATION_NOT_ACCEPTED
  This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

1. This call does not change the *security_user_ID* for the specified conversation.
2. CPI Communications returns the *security_user_ID* using the native encoding of the local system.
3. The call is not associated with any conversation queue. When a conversation uses queue-level non-blocking, the call does not return CM_OPERATION_NOT_ACCEPTED on the conversation.

Related Information

"Automatic Conversion of Characteristics" on page 41 provides further information on the automatic conversion of the *security_user_ID* parameter.

"Set_Conversation_Security_User_ID (CMSCSU)" on page 298 discusses the setting of the *security_user_ID* characteristic.

Extract_Send_Receive_Mode (CMESRM)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
				X	X*			

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

The *send_receive_mode* conversation characteristic indicates whether the conversation is using half-duplex or full-duplex mode for data transmission.

X* In OS/2, this call is supported by Communications Server.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

send_receive_mode *(output)*
Specifies the send-receive mode for the conversation.

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

- CM_HALF_DUPLEX
  Indicates that the conversation is a half-duplex conversation.
- CM_FULL_DUPLEX
  Indicates that the conversation is a full-duplex conversation.

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

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the *conversation_ID* specifies a conversation in **Initialize-Incoming** state.

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

State Changes

This call does not cause a state change.

Usage Notes

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

Related Information

"Send-Receive Modes" on page 19 provides more information on the differences between half-duplex and full-duplex conversations.

"Example 8: Establishing a Full-Duplex Conversation" on page 84 shows an example of how a full-duplex conversation is set up.

"Set_Send_Receive_Mode (CMSSRM)" on page 349 describes how to set the *send_receive_mode* characteristic for a conversation.

Extract_Sync_Level (CMESL)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

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

The *sync_level* conversation characteristic indicates the level of synchronization services provided on the conversation.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

sync_level *(output)*
Specifies the variable containing the *sync_level* characteristic of this conversation. The *sync_level* variable can have one of the following values:

- CM_NONE
  Specifies that the programs will not perform confirmation processing on this conversation. The programs will not issue calls or recognize returned parameters relating to synchronization.
- CM_CONFIRM (half-duplex conversations only)
  Specifies that the programs can perform confirmation processing on this conversation. The programs can issue calls and will recognize returned parameters relating to confirmation.
- CM_SYNC_POINT (half-duplex conversations only)
  For systems that support resource recovery processing, this value specifies that this conversation is a protected resource. The programs can issue resource recovery calls and will recognize returned parameters relating to resource recovery operations. The programs can also perform confirmation processing.

  The CM_SYNC_POINT value is not returned on AIX, IMS, MVS, NS/WIN, OS/2, or Windows 95 systems.
- CM_SYNC_POINT_NO_CONFIRM
  For systems that support resource recovery processing, this value specifies that the conversation is a protected resource. The programs can issue resource recovery interface calls and will recognize returned parameters relating to resource recovery processing. The programs cannot perform confirmation processing.

Notes:

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

return_code *(output)*

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

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the *conversation_ID* specifies a conversation in **Initialize-Incoming** state.
- CM_OPERATION_NOT_ACCEPTED
  This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

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

Related Information

"Set_Sync_Level (CMSSL)" on page 354 provides more information on the *sync_level* characteristic.

Extract_TP_Name (CMETPN)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X*				X	X*		X*	X

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

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

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

X* In Aix, this call is supported in Version 3 Release 1 or later.
In OS/2, this call is supported by Communications Server.
VM supports this function via a product-specific extension call.
For more information, see “Extract_TP_Name (XCETPN)” on page 554.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

TP_name *(output)*
Specifies the variable containing the *TP_name* for the specified conversation. The variable must be defined with a length of at least 64 bytes.

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

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

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

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is in **Initialize-Incoming** state.
- CM_OPERATION_NOT_ACCEPTED
  This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

1. This call is used by programs that accept multiple conversations. Extract_TP_Name allows the program to determine which local name was specified in the incoming conversation startup request.
2. The call is not associated with any conversation queue. When a conversation uses queue-level non-blocking, the call does not return CM_OPERATION_NOT_ACCEPTED on the conversation.
3. TP_Name is returned using the local system's native encoding. CPI-C automatically converts TP_Name from the transfer encoding where necessary.
4. Refer to "SNA Service Transaction Programs" on page 727 for special handling of SNA Service Transaction Program names.

Related Information

"Specify_Local_TP_Name (CMSLTP)" on page 361 provides more information on handling multiple names within a single program.

Extract_Transaction_Control (CMETC)

OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32

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

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

The *transaction_control* conversation characteristic indicates whether the conversation supports chained or unchained transactions.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

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

- CM_CHAINED_TRANSACTIONS
  Specifies that the conversation uses chained transactions.
- CM_UNCHAINED_TRANSACTIONS
  Specifies that the conversation uses unchained transactions.

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

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

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

- CM_PROGRAM_STATE_CHECK
  This value indicates that the *conversation_ID* specifies a conversation in **Initialize-Incoming** state.
- CM_OPERATION_NOT_ACCEPTED
  This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

1. This call does not change the *transaction_control* for the specified conversation.
2. This call is not associated with any conversation queue. When a conversation uses queue-level non-blocking, the call does not return CM_OPERATION_NOT_ACCEPTED.
3. This call can be used by the recipient to determine the *transaction_control* characteristic for the conversation. If the value is CM_CHAINED_TRANSACTIONS, the conversation is already included in a transaction. If the value is CM_UNCHAINED_TRANSACTIONS, the program will be informed with a CM_JOIN_TRANSACTION *status_received* value if it is to join the transaction.

Related Information

"Set_Transaction_Control (CMSTC)" on page 359 provides more information on the *transaction_control* characteristic.

Flush (CMFLUS)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

A program uses the Flush (CMFLUS) call to empty the local system's send buffer for a given conversation. When notified by CPI Communications that a Flush has been issued, the system sends any information it has buffered to the remote system. The information that can be buffered comes from the Allocate, Send_Data, or Send_Error call. Refer to the descriptions of these calls for more details of when and how buffering occurs.

Format

```
CALL CMFLUS(conversation_ID,
            return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

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

- CM_OK
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates that the *conversation_ID* specifies an unassigned conversation ID.
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  – For a half-duplex conversation, the conversation is not in **Send**, **Send-Pending**, or **Defer-Receive** state.
  – For a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the conversation's context is in the **Backout-Required** condition. The Flush call is not allowed for this conversation while its context is in this condition.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

The following return codes apply to full-duplex conversations.

If the conversation is not currently included in a transaction, the *return_code* can have one of the following values:

- CM_ALLOCATION_ERROR
- CM_DEALLOCATED_ABEND
- CM_DEALLOCATED_ABEND_SVC
- CM_DEALLOCATED_ABEND_TIMER
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_DEALLOCATED_NORMAL
- CM_PROGRAM_STATE_CHECK
  This value indicates that the program has received a *status_received* value of CM_JOIN_TRANSACTION and must issue a tx_begin call to the X/Open TX interface to join the transaction.

If the *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM and the conversation is included in a transaction, the *return_code* can have one of the following values:

- CM_TAKE_BACKOUT
- CM_DEALLOCATED_ABEND_BO
- CM_DEALLOCATED_ABEND_SVC_BO
- CM_DEALLOCATED_ABEND_TIMER_BO
- CM_RESOURCE_FAILURE_RETRY_BO
- CM_RESOURCE_FAIL_NO_RETRY_BO
- CM_INCLUDE_PARTNER_REJECT_BO
- CM_CONV_DEALLOC_AFTER_SYNCPT

State Changes

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

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

For full-duplex conversations, this call does not cause any state changes.

Usage Notes

1. This call optimizes processing between the local and remote programs. The local system normally buffers the data from consecutive Send_Data calls until it has a sufficient amount for transmission. Only then does the local system transmit the buffered data.

   The local program can issue a Flush call to cause the system to transmit the data immediately. This helps minimize any delay in the remote program's processing of the data.
2. The Flush call causes the local system to flush its send buffer only when the system has some information to transmit. If the system has no information in its send buffer, nothing is transmitted to the remote system.
3. The use of Send_Data followed by a call to Flush is equivalent to the use of Send_Data after setting *send_type* to CM_SEND_AND_FLUSH.
4. For full-duplex conversations, when CM_ALLOCATION_ERROR, CM_DEALLOCATE_*, CM_RESOURCE_FAILURE_*, or CM_DEALLOCATE_NORMAL

is returned and the conversation is in **Send-Receive** state, the program can terminate the conversation by issuing Receive calls until it gets a return code that takes it to **Reset** state, or by issuing a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND.

Related Information

"Data Buffering and Transmission" on page 44 discusses the conditions for data transmission.

"Example 5: Validation of Data Receipt" on page 78 shows an example of how a program can use the Flush call to establish a conversation immediately.

"Allocate (CMALLC)" on page 124 provides more information on how information is buffered from the Allocate call.

"Send_Data (CMSEND)" on page 249 provides more information on how information is buffered from the Send_Data call.

"Send_Error (CMSERR)" on page 259 provides more information on how information is buffered from the Send_Error call.

"Send_Mapped_Data (CMSNDM)" on page 271 provides more information on a program sends mapped data to its partner.

"Set_Send_Type (CMSST)" on page 351 discusses alternative methods of achieving the Flush function.

Include_Partner_In_Transaction (CMINCL)

OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32

A program uses the Include_Partner_In_Transaction (CMINCL) call to explicitly request that the subordinate join the transaction. The caller must be in a transaction, and the subordinate must be on a branch supporting unchained transactions.

Note: The Include_Partner_In_Transaction call has meaning only when an OSI TP CRM is being used for the conversation.

Format

```
CALL CMINCL(conversation_ID,
            return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  – The *conversation_ID* specifies an unassigned conversation identifier.
  – The conversation is not using an OSI TP CRM.
  – The *transaction_control* is CM_CHAINED_TRANSACTIONS.
  – The program is not the superior for the conversation.
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  – For a half-duplex conversation, the conversation is not in **Send** or **Send-Pending** state.
  – For a full-duplex conversation, the conversation is not in **Send-Receive** state.
  – The conversation is basic and the program started but did not finish sending a logical record.
  – The conversation's context is not in transaction. The program must issue a tx_begin call to the X/Open TX interface to start a transaction.
  – The conversation is already included in the current transaction.
- CM_DEALLOCATED_ABEND
- CM_RESOURCE_FAILURE_NO_RETRY

- CM_RESOURCE_FAILURE_RETRY
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

The following return codes apply to half-duplex conversations.

- CM_SYNC_LVL_NOT_SUPPORTED_SYS
- CM_SEND_RCV_MODE_NOT_SUPPORTED
- CM_TPN_NOT_RECOGNIZED
- CM_TP_NOT_AVAILABLE_NO_RETRY
- CM_TP_NOT_AVAILABLE_RETRY
- CM_PROGRAM_ERROR_PURGING

The following return codes apply to full-duplex conversations.

- CM_ALLOCATION_ERROR
- CM_DEALLOCATED_NORMAL

State Changes

This call does not cause any state changes.

Usage Notes

1. The call is used by a program to request that the subordinate join the transaction when the *begin_transaction* conversation characteristic set to CM_BEGIN_EXPLICIT.
2. The remote program receives the request to join the transaction as a *status_received* indicator of CM_JOIN_TRANSACTION on a Receive call.

Related Information

"Chained and Unchained Transactions" on page 61 discusses chained and unchained transactions.

"Joining a Transaction" on page 61 discusses how a program requests the partner program to join a transaction.

"Set_Begin_Transaction (CMSBT)" on page 288 discusses the *begin_transaction* characteristic.

Initialize_Conversation (CMINIT)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

A program uses the Initialize_Conversation (CMINIT) call to initialize values for various conversation characteristics before the conversation is allocated (with a call to Allocate). The remote partner program uses the Accept_Conversation call or the Initalize_Incoming and Accept_Incoming calls to initialize values for the conversation characteristics on its end of the conversation.

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

Format

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

Parameters

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

sym_dest_name *(input)*
Specifies the symbolic destination name. The symbolic destination name is provided by the program and points to an entry in the side information. The appropriate entry in the side information is retrieved and used to initialize the characteristics for the conversation. Alternatively, a blank *sym_dest_name* (one composed of eight space characters) may be specified. When this is done, the program is responsible for setting up the appropriate destination information, using Set calls, before issuing the Allocate call for that conversation.

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

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

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

On VM, if the value specified for the *sym_dest_name* does not match an entry in side information, VM/ESA does not return CM_PROGRAM_PARAMETER_CHECK. Instead, CM_OK is returned and the specified *sym_dest_name* is used to set the TP_name characteristic for the conversation. See Chapter 12, “CPI Communications on VM/ESA CMS” on page 527 for more information.

- CM_PRODUCT_SPECIFIC_ERROR

State Changes

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

Usage Notes

1. For a list of the conversation characteristics that are initialized when the Initialize_Conversation call completes successfully, see Table 3 on page 35.
2. For each conversation, CPI Communications assigns a unique identifier, the *conversation_ID*. The program then uses the *conversation_ID* in all future calls intended for that conversation. Initialize_Conversation (or Accept_Conversation or Initialize_For_Incoming, on the opposite side of the conversation) must be issued by the program before any other calls may be made for that conversation.
3. A program can call Initialize_Conversation more than once and establish multiple, concurrently active conversations. When a program with an existing initialized conversation issues an Initialize_Conversation call, CPI Communications initializes a new conversation and assigns a new *conversation_ID*. CPI Communications is designed so that Initialize_Conversation is always issued from the **Reset** state. For more information about managing concurrent conversations, see “Multiple Conversations” on page 30.
4. If the side information supplies invalid allocation information on the Initialize_Conversation (CMINIT) call, or if the program supplies invalid allocation information on any subsequent Set calls, the error is detected when the information is processed by Allocate (CMALLC).
5. A program may obtain information about its partner program (for example, *partner_LU_name*, *TP_name*, and *mode_name*) from a source other than the side information. The local program can, for example, read this information from a file or receive it from another partner over a separate conversation. The information might even be hard-coded in the program. In cases where a program wishes to specify destination information about its partner program without making use of side information, the local program may supply a blank *sym_dest_name* on the Initialize_Conversation call. CPI Communications will initialize the conversation characteristics and return a *conversation_ID* for the new conversation. The program is then responsible for specifying valid destination information (using Set_Partner_LU_Name, Set_TP_Name, and Set_Mode_Name calls) before issuing the Allocate call.

Related Information

"Side Information" on page 23 provides more information on *sym_dest_name*.

"Conversation Characteristics" on page 33 provides a general overview of conversation characteristics and how they are used by the program and CPI Communications.

"Example 1: Data Flow in One Direction" on page 69 shows an example program flow where Initialize_Conversation is used.

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

Initialize_For_Incoming (CMINIC)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X*				X	X*			X

A program uses the Initialize_For_Incoming and Accept_Incoming calls, rather than the Accept call, when various conversation characteristics (for example, processing_mode) must be initialized before the conversation is accepted.

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

X* In AIX, this call is supported in Version 3 Release 1 or later. In OS/2, this call is supported by Communications Server.

Format

CALL CMINIC(*conversation_ID*,
return_code)

Parameters

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

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

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

Usage Notes

1. For a list of the conversation characteristics initialized when the Initialize_For_Incoming call completes successfully, see Table 3 on page 35.
2. For each conversation, CPI Communications assigns a unique identifier, the *conversation_ID*. The program then uses the *conversation_ID* in all future calls intended for that conversation.

3. The call is designed for use with Accept_Incoming. As shown in Table 3, when Initialize_For_Incoming completes, certain conversation characteristics are initialized. When the Accept_Incoming call completes, the remaining applicable conversation characteristics are initialized.
4. The Initialize_For_Incoming and Accept_Incoming calls can be used by a program to accept multiple conversations.
5. Even though the program may have been initialized as a direct result of an incoming conversation initialization request, the conversation must not be coupled with the *conversation_id* until the *Accept_Incoming* call. This allows the program to issue *Set_TP_Name* calls between the *Initialize_For_Incoming* and *Accept_Incoming* calls to associate specific transaction program names with specific *conversation_ids*.

Related Information

“Conversation Characteristics” on page 33 describes how the conversation characteristics are initialized by the Initialize_For_Incoming and Accept_Incoming calls.

“Example 12: Accepting Multiple Conversations Using Blocking Calls” on page 92 and “Example 13: Accepting Multiple Conversations Using Conversation-Level Non-Blocking Calls” on page 94 show example program flows where Initialize_For_Incoming is used.

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

“Accept_Incoming (CMACCI)” on page 121 describes how the *conversation_ID* is used when an incoming conversation is accepted by a program.

Prepare (CMPREP)

OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32

A program uses the Prepare (CMPREP) call to explicitly request that a branch of the transaction prepare its resources to commit changes made during the transaction.

When the conversation is using an OSI TP CRM and the caller is not the root of the transaction, the caller must have received a take-commit notification from its superior. The subordinate program on the conversation cannot issue a Prepare (CMPREP) call.

Format

```
CALL CMPREP(conversation_ID,
            return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *sync_level* is not CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.
  - The conversation is using an OSI TP CRM, and the program is not the superior for the conversation.
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - For a half-duplex conversation, the conversation is not in **Send**, **Send-Pending**, **Defer-Receive**, or **Defer-Deallocate** state.
  - For a full-duplex conversation, the conversation is not in **Send-Receive** or **Defer-Deallocate** state.
  - The conversation is basic, and the program started but did not finish sending a logical record.
  - The conversation's context is not in transaction. The program must issue a tx_begin call to the X/Open TX interface to start a transaction.

- The *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and the conversation's context is in the **Backout-Required** condition.
- The conversation is using an OSI TP CRM, *begin_transaction* is set to CM_BEGIN_EXPLICIT, and the conversation is not currently included in a transaction.
- The conversation is using an OSI TP CRM, and the program is not the root of the transaction and has not received a take-commit notification from its superior.

- CM_TAKE_BACKOUT
- CM_DEALLOCATED_ABEND_BO
- CM_DEALLOCATED_ABEND_SVC_BO
- CM_DEALLOCATED_ABEND_TIMER_BO
- CM_RESOURCE_FAILURE_RETRY_BO
- CM_RESOURCE_FAIL_NO_RETRY_BO
- CM_INCLUDE_PARTNER_REJECT_BO
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

The following return codes apply to half-duplex conversations.

- CM_CONVERSATION_TYPE_MISMATCH
- CM_PIP_NOT_SPECIFIED_CORRECTLY
- CM_SECURITY_NOT_VALID
- CM_SYNC_LVL_NOT_SUPPORTED_PGM
- CM_SYNC_LVL_NOT_SUPPORTED_SYS
- CM_SEND_RCV_MODE_NOT_SUPPORTED
- CM_TPN_NOT_RECOGNIZED
- CM_TP_NOT_AVAILABLE_NO_RETRY
- CM_TP_NOT_AVAILABLE_RETRY
- CM_PROGRAM_ERROR_PURGING

The following return codes apply to full-duplex conversations.

- CM_ALLOCATION_ERROR
- CM_CONV_DEALLOC_AFTER_SYNCPT

State Changes

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

Usage Notes

1. The Prepare (CMPREP) functions without waiting for a response from the remote program. A program can detect that its partner has prepared its transaction resources by issuing a Receive (CMRCV) call and checking the result in *status_received*, or the caller can complete the transaction without waiting for the partner to respond.
2. A program cannot send data to the remote program after issuing a Prepare call.

3. A program that issues the Prepare call may receive data from the remote program if the conversation is using an OSI TP CRM and either *prepare_data_permitted* is set to CM_PREPARE_DATA_PERMITTED or the conversation is full-duplex.
4. The partner finds out about the Prepare call by receiving one of the take-commit notifications described in Table 9 on page 56 or Table 10 on page 57.

Related Information

"Receive (CMRCV)" on page 213 discusses the *status_received* parameter.

"Set_Prepare_Data_Permitted (CMSPDP)" on page 329 discusses the *prepare_data_permitted* characteristic.

Prepare_To_Receive (CMPTR)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

Programs use the Prepare_To_Receive call to transfer control of the conversation to the remote program without waiting for the remote program to send data; that is, waiting for a Receive to complete. This allows the local program to continue processing until it actually needs data from the remote program.

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

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

Call CMSPTR – Set_Prepare_To_Receive_Type

Note: The Prepare_To_Receive_Type call can be issued only on a half-duplex conversation.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

return_code *(output)*
Specifies the result of the call execution. The *prepare_to_receive_type* currently in effect determines which return codes can be returned to the local program.

For any of the following conditions:

- *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_FLUSH
- *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* is set to CM_NONE
- *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, but the conversation is not currently included in a transaction

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

- CM_OK
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  – The conversation is not in **Send** or **Send-Pending** state.
  – The conversation is basic and in **Send** state, and the program started but did not finish sending a logical record.
  – The *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and the conversation's context is in the **Backout-Required** condition. The Prepare_To_Receive call is not allowed while the conversation's context is in this condition.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  – The *conversation_ID* specifies an unassigned conversation identifier.
  – The *send_receive_mode* of the conversation is CM_FULL_DUPLEX.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

For any of the following conditions:

– *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_CONFIRM
– *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* is set to CM_CONFIRM
– *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* is set to CM_SYNC_POINT, but the conversation is not currently included in a transaction

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

- CM_OK
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_CONVERSATION_TYPE_MISMATCH
- CM_PIP_NOT_SPECIFIED_CORRECTLY
- CM_SECURITY_NOT_VALID
- CM_SYNC_LVL_NOT_SUPPORTED_PGM
- CM_SYNC_LVL_NOT_SUPPORTED_SYS
- CM_SEND_RCV_MODE_NOT_SUPPORTED
- CM_TPN_NOT_RECOGNIZED
- CM_TP_NOT_AVAILABLE_NO_RETRY
- CM_TP_NOT_AVAILABLE_RETRY
- CM_DEALLOCATED_ABEND
- CM_PROGRAM_ERROR_PURGING
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_DEALLOCATED_ABEND_SVC (basic conversations only)
- CM_DEALLOCATED_ABEND_TIMER (basic conversations only)
- CM_SVC_ERROR_PURGING (basic conversations only)
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  – The conversation is not in **Send** or **Send-Pending** state.
  – The conversation is basic and in **Send** state, and the program started but did not finish sending a logical record.

  - For a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the conversation's context is in the **Backout-Required** condition.
- CM_PROGRAM_PARAMETER_CHECK
  This return code indicates that the *conversation_ID* specifies an unassigned conversation identifier.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR
- The following values are returned only when *sync_level* is set to CM_SYNC_POINT:
  - CM_TAKE_BACKOUT
  - CM_DEALLOCATED_ABEND_BO
  - CM_DEALLOCATED_ABEND_SVC_BO (basic conversations only)
  - CM_DEALLOCATED_ABEND_TIMER_BO (basic conversations only)
  - CM_RESOURCE_FAIL_NO_RETRY_BO
  - CM_RESOURCE_FAILURE_RETRY_BO
  - CM_INCLUDE_PARTNER_REJECT_BO

If *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, *sync_level* is set to is CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and the conversation is included in a transaction, the *return_code* variable can have one of the following values:

- CM_OK
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The conversation is not in **Send** or **Send-Pending** state.
  - The conversation is basic and in **Send** state, and the program started but did not finish sending a logical record.
  - The conversation's context is in the **Backout-Required** condition.
  - A prior call to Deferred_Deallocate is still in effect for the conversation.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The conversation is using an OSI TP CRM, and the program is not the superior for the conversation.
  - The *send_receive_mode* of the conversation is CM_FULL_DUPLEX.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

When *return_code* indicates CM_OK:

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

or CM_SYNC_POINT_NO_CONFIRM, but the conversation is not currently included in a transaction.

- The conversation enters the **Defer-Receive** state if *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and the conversation is included in a transaction.

Usage Notes

1. If *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_CONFIRM, or if *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* is CM_CONFIRM, or if *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* is CM_SYNC_POINT but the conversation is not currently included in a transaction, the local program regains control when a Confirmed reply is received.

2. The program uses the *prepare_to_receive_type* characteristic set to CM_PREP_TO_RECEIVE_SYNC_LEVEL to transfer send control to the remote program based on one of the following synchronization levels allocated to the conversation:

   - If *sync_level* is set to CM_NONE, or if *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM but the conversation is not currently included in a transaction, the system's send buffer is flushed if it contains information, and send control is transferred to the remote program without any synchronizing acknowledgment.

   - If *sync_level* is set to CM_CONFIRM, or if *sync_level* is set to CM_SYNC_POINT but the conversation is not currently included in a transaction, the system's send buffer is flushed if it contains information, and send control is transferred to the remote program with confirmation requested.

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

3. The program uses the *prepare_to_receive_type* characteristic set to CM_PREP_TO_RECEIVE_FLUSH to transfer send control to the remote program without any synchronizing acknowledgment. The *prepare_to_receive_type* characteristic set to CM_PREP_TO_RECEIVE_FLUSH functions the same as the *prepare_to_receive_type* characteristic set to CM_PREP_TO_RECEIVE_SYNC_LEVEL combined with a *sync_level* set to CM_NONE.

4. The program uses the *prepare_to_receive_type* characteristic set to CM_PREP_TO_RECEIVE_CONFIRM to transfer send control to the remote program with confirmation requested. The *prepare_to_receive_type* characteristic set to CM_PREP_TO_RECEIVE_CONFIRM functions the same as the *prepare_to_receive_type* characteristic set to CM_PREP_TO_RECEIVE_SYNC_LEVEL combined with a *sync_level* set to CM_CONFIRM.

5. The remote program receives send control of the conversation by means of the *status_received* parameter, which can have the following values:
   - CM_SEND_RECEIVED
     The local program issued this call with one of the following:
     - *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_FLUSH
     - *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_NONE
     - *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT_NO_CONFIRM, but with the conversation not currently included in a transaction.
   - CM_CONFIRM_SEND_RECEIVED
     The local program issued this call with one of the following:
     - *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_CONFIRM
     - *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_CONFIRM
     - *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT, but with the conversation not currently included in a transaction.
   - CM_TAKE_COMMIT_SEND
     The local program issued a resource recovery commit call after issuing the Prepare_To_Receive call with *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and with the conversation included in a transaction.
6. When the local program's end of the conversation enters **Receive** state, the remote program's end of the conversation enters **Send** or **Send-Pending** state, depending on the *data_received* indicator. The remote program can then send data to the local program.
7. When the conversation is using an OSI TP CRM and the Deferred_Deallocate call has been issued on the conversation, the Prepare_To_Receive call with *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL is not allowed. The call gets the CM_PROGRAM_PARAMETER_CHECK return code.

Related Information

"Example 3: The Sending Program Changes the Data Flow Direction" on page 74 and "Example 4: The Receiving Program Changes the Data Flow Direction" on page 75 show example program flows where the Prepare_To_Receive call is used.

"Set_Confirmation_Urgency (CMSCU)" on page 290 tells how to request an immediate response to the Prepare_To_Receive call.
"Set_Prepare_To_Receive_Type (CMSPTR)" on page 331 provides more information on the *prepare_to_receive_type* characteristic.

"Set_Sync_Level (CMSSL)" on page 354 discusses the *sync_level* characteristic and its possible values.

Receive (CMRCV)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

A program uses the Receive (CMRCV) call to receive information from a given conversation. The information received can be a data record (on a mapped conversation), data (on a basic conversation), conversation status, or a request for confirmation or for resource recovery services.

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

CALL CMSF — Set_Fill
CALL CMSRT — Set_Receive_Type

Format

```
CALL CMRCV(conversation_ID,
           buffer,
           requested_length,
           data_received,
           received_length,
           status_received,
           control_information_received,
           return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

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

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

requested_length *(input)*
Specifies the maximum amount of data the program is to receive. Valid *requested_length* values range from 0 to the maximum buffer size supported by the system. The maximum buffer size is at least 32767 bytes. See Usage Note 13 on page 225 for additional information about determining the maximum buffer size.

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

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

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

- CM_NO_DATA_RECEIVED (basic and mapped conversations)
  No data is received by the program. Status may be received if the *return_code* is set to CM_OK.
- CM_DATA_RECEIVED (basic conversations only)
  The *fill* characteristic is set to CM_FILL_BUFFER and data (independent of its logical-record format) is received by the program.
- CM_COMPLETE_DATA_RECEIVED (basic and mapped conversations)
  This value indicates one of the following:
  - For mapped conversations, a complete data record or the last remaining portion of the record is received.
  - For basic conversations, *fill* is set to CM_FILL_LL and a complete logical record, or the last remaining portion of the record, is received.
- CM_INCOMPLETE_DATA_RECEIVED (basic and mapped conversations)
  This value indicates one of the following:
  - For mapped conversations, less than a complete data record is received.
  - For basic conversations, *fill* is set to CM_FILL_LL, and less than a complete logical record is received.

  Note: For either type of conversation, if *data_received* is set to CM_INCOMPLETE_DATA_RECEIVED, the program must issue another Receive (or possibly multiple Receive calls) to receive the remainder of the data.

received_length *(output)*
Specifies the variable containing the amount of data the program received, up to the maximum. If the program does not receive data on this call, the value contained in *received_length* has no meaning.

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

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

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

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

- CM_NO_STATUS_RECEIVED
  No conversation status is received by the program; data may be received.
- CM_SEND_RECEIVED (half-duplex conversations only)
  The remote program's end of the conversation has entered **Receive** state, placing the local program's end of the conversation in **Send-Pending** state (if the program also received data on this call) or **Send** state (if the program did not receive data on this call). The local program (which issued the Receive call) can now send data.
- CM_CONFIRM_RECEIVED (half-duplex conversations only)
  The remote program has sent a confirmation request, requesting the local program to respond by issuing a Confirmed call. The local program must respond by issuing Confirmed, Send_Error, Deallocate with *deallocate_type* set to CM_DEALLOCATE_ABEND, or Cancel_Conversation.

- CM_CONFIRM_SEND_RECEIVED (half-duplex conversations only)
  The remote program's end of the conversation has entered **Receive** state with confirmation requested. The local program must respond by issuing Confirmed, Send_Error, Deallocate with *deallocate_type* set to CM_DEALLOCATE_ABEND, or Cancel_Conversation. Upon issuing a successful Confirmed call, the local program (which issued the Receive call) can now send data.
- CM_CONFIRM_DEALLOC_RECEIVED
  The remote program has deallocated the conversation with confirmation requested. The local program must respond by issuing Confirmed, Send_Error, Deallocate with *deallocate_type* set to CM_DEALLOCATE_ABEND, or Cancel_Conversation. Upon issuing a successful Confirmed call, the local program (which issued the Receive call) is deallocated—that is, placed in **Reset** state.

For a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and with the conversation included in a transaction, the *status_received* variable can also be set to one of the following values:

- CM_TAKE_COMMIT
  The remote program has issued a resource recovery commit call or a Prepare call. For the exact conditions for receipt of this *status_received* value, refer to Table 9 on page 56 and Table 10 on page 57. The local program should issue a commit call in order to commit all protected resources throughout the transaction. When appropriate, the local program may respond by issuing a call other than commit, such as Send_Error (for half-duplex conversations only) or a resource recovery backout call.
- CM_TAKE_COMMIT_SEND (for half-duplex conversations only)
  The remote program has issued a Prepare_To_Receive call with *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and then issued a resource recovery commit call or a Prepare call. For the exact conditions for receipt of this *status_received* value, refer to Table 9 on page 56 and Table 10 on page 57. The local program should issue a commit call in order to commit all protected resources throughout the transaction. When appropriate, the local program may respond by issuing a call other than commit, such as Send_Error or a resource recovery backout call. If a successful commit call is issued, the local program can then send data.
- CM_TAKE_COMMIT_DEALLOCATE
  The remote program has deallocated the conversation with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, or has issued a Deferred_Deallocate call, and then issued a resource recovery commit call or a Prepare call. For the exact conditions for receipt of this *status_received* value, refer to Table 9 on page 56 and Table 10 on page 57. The local program should issue a commit call in order to commit all protected resources throughout the transaction. The local program may respond by issuing a call other than commit when appropriate, such as Send_Error for a half-duplex conversation, or a resource recovery backout call. If a successful commit call is issued, the local program is then deallocated—that is, placed in **Reset** state.

- CM_PREPARE_OK
  By issuing a Prepare (CMPREP) call, the local program requested that the remote program prepare its resources for commitment, and the remote program has done so, by issuing a commit call. The subtree is now ready to commit its resources.

For a conversation with *sync_level* set to either CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and using an OSI TP CRM, the *status_received* variable can also be set to one of the following values if the conversation is included in a transaction:

- CM_TAKE_COMMIT_DATA_OK
  The remote program issued a Prepare call. For the exact conditions for receipt of this *status_received* value, refer to Table 9 on page 56 and Table 10 on page 57. The local program should issue a commit call in order to commit all protected resources throughout the transaction. The program is allowed to send data before issuing the commit call. When appropriate, the local program may respond by issuing a call other than commit, such as Send_Error (for half-duplex conversations only) or a resource recovery backout call.
- CM_TAKE_COMMIT_SEND_DATA_OK (half-duplex conversations only)
  The remote program issued a Prepare call. For the exact conditions for receipt of this *status_received* value, refer to Table 9 on page 56 and Table 10 on page 57. The local program should issue a commit call in order to commit all protected resources throughout the transaction. The program is allowed to send data before issuing the commit call. When appropriate, the local program may respond by issuing a call other than commit, such as Send_Error or a resource recovery backout call. If a successful commit call is issued, the local program can then send data.
- CM_TAKE_COMMIT_DEALLOC_DATA_OK
  The remote program issued a Prepare call. For the exact conditions for receipt of this *status_received* value, refer to Table 9 on page 56 and Table 10 on page 57. The local program should issue a commit call in order to commit all protected resources throughout the transaction. The program is allowed to send data before issuing the commit call. The local program may respond by issuing a call other than commit when appropriate, such as Send_Error for a half-duplex conversation, or a resource recovery backout call. If a successful commit call is issued, the local program is then deallocated—that is, placed in **Reset** state.
- CM_JOIN_TRANSACTION (unchained transactions only)
  The remote program requested that the local program join into its current transaction. If the local program called Set_Join_Transaction with CM_JOIN_EXPLICIT, the local program should issue a tx_begin() call to the X/Open TX interface to join the superior's transaction as soon as any local work, which is not to be included in the remote program's transaction, has completed. If the local program called Set_Join_Transaction with CM_JOIN_IMPLICIT, the local program already joined the transaction, and tx_begin() must not be called.

control_information_received *(output)*
Specifies the variable containing an indication of whether or not control information has been received. The *control_information_received* variable can have one of the following values:

- CM_NO_CONTROL_INFO_RECEIVED
  Indicates that no control information was received.

- CM_REQ_TO_SEND_RECEIVED (half-duplex conversations only)
  The local program received a request-to-send notification from the remote program. The remote program issued Request_To_Send, requesting the local program's end of the conversation to enter **Receive** state, which would place the remote program's end of the conversation in **Send** state. See “Request_To_Send (CMRTS)” on page 246 for further discussion of the local program's possible responses.
- CM_ALLOCATE_CONFIRMED (OSI TP CRM only)
  The local program received confirmation of the remote program's acceptance of the conversation.
- CM_ALLOCATE_CONFIRMED_WITH_DATA (OSI TP CRM only)
  The local program received confirmation of the remote program's acceptance of the conversation. The local program may now issue an Extract_Initialization_Data (CMEID) call to receive the initialization data.
- CM_ALLOCATE_REJECTED_WITH_DATA (OSI TP CRM only)
  The remote program rejected the conversation. The local program may now issue an Extract_Initialization_Data (CMEID) call to receive the initialization data.

  This value will be returned with a return code of CM_OK. The program will receive a CM_DEALLOCATED_ABEND return code on a later call on the conversation.
- CM_EXPEDITED_DATA_AVAILABLE (LU 6.2 CRM only)
  Expedited data is available to be received.
- CM_RTS_RCVD_AND_EXP_DATA_AVAIL (half-duplex conversations and LU 6.2 CRM only)
  The local program received a request-to-send notification from the remote program and expedited data is available to be received.

Notes:

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

return_code *(output)*
Specifies the result of the call execution. The return codes that can be returned depend on the state and characteristics of the conversation at the time this call is issued.

***The following return codes apply to half-duplex conversations*.**

If *receive_type* is set to CM_RECEIVE_AND_WAIT and this call is issued in **Send** state, *return_code* can have one of the following values:

- CM_OK

- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_CONVERSATION_TYPE_MISMATCH
- CM_PIP_NOT_SPECIFIED_CORRECTLY
- CM_SECURITY_NOT_VALID
- CM_SYNC_LVL_NOT_SUPPORTED_PGM
- CM_SYNC_LVL_NOT_SUPPORTED_SYS
- CM_SEND_RCV_MODE_NOT_SUPPORTED
- CM_TPN_NOT_RECOGNIZED
- CM_TP_NOT_AVAILABLE_NO_RETRY
- CM_TP_NOT_AVAILABLE_RETRY
- CM_DEALLOCATED_ABEND
- CM_DEALLOCATED_NORMAL
- CM_PROGRAM_ERROR_NO_TRUNC
- CM_PROGRAM_ERROR_PURGING
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_DEALLOCATED_ABEND_SVC (basic conversations only)
- CM_DEALLOCATED_ABEND_TIMER (basic conversations only)
- CM_SVC_ERROR_NO_TRUNC (basic conversations only)
- CM_SVC_ERROR_PURGING (basic conversations only)
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR
- The following values are returned only when *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and the conversation is included in a transaction :
  - CM_TAKE_BACKOUT
  - CM_DEALLOCATED_ABEND_BO
  - CM_DEALLOCATED_ABEND_SVC_BO (basic conversations only)
  - CM_DEALLOCATED_ABEND_TIMER_BO (basic conversations only)
  - CM_RESOURCE_FAIL_NO_RETRY_BO
  - CM_RESOURCE_FAILURE_RETRY_BO
  - CM_INCLUDE_PARTNER_REJECT_BO

If *receive_type* is set to CM_RECEIVE_AND_WAIT and this call is issued in **Send-Pending** state, *return_code* can be one of the following values:

- CM_OK
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_DEALLOCATED_ABEND
- CM_DEALLOCATED_NORMAL
- CM_PROGRAM_ERROR_NO_TRUNC
- CM_PROGRAM_ERROR_PURGING
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_DEALLOCATED_ABEND_SVC (basic conversations only)
- CM_DEALLOCATED_ABEND_TIMER (basic conversations only)
- CM_SVC_ERROR_NO_TRUNC (basic conversations only)
- CM_SVC_ERROR_PURGING (basic conversations only)
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR
- The following values are returned only when *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and the conversation is included in a transaction :

- CM_TAKE_BACKOUT
- CM_DEALLOCATED_ABEND_BO
- CM_DEALLOCATED_ABEND_SVC_BO (basic conversations only)
- CM_DEALLOCATED_ABEND_TIMER_BO (basic conversations only)
- CM_RESOURCE_FAIL_NO_RETRY_BO
- CM_RESOURCE_FAILURE_RETRY_BO
- CM_INCLUDE_PARTNER_REJECT_BO

If *receive_type* is set to CM_RECEIVE_AND_WAIT or CM_RECEIVE_IMMEDIATE and this call is issued in **Receive** or **Prepared** state, *return_code* can be one of the following:

- CM_OK
- CM_OPERATION_INCOMPLETE
  This value is only received when *receive_type* is set to CM_RECEIVE_AND_WAIT.
- CM_CONVERSATION_CANCELLED
- CM_CONVERSATION_TYPE_MISMATCH
- CM_PIP_NOT_SPECIFIED_CORRECTLY
- CM_SECURITY_NOT_VALID
- CM_SYNC_LVL_NOT_SUPPORTED_PGM
- CM_SYNC_LVL_NOT_SUPPORTED_SYS
- CM_SEND_RCV_MODE_NOT_SUPPORTED
- CM_TPN_NOT_RECOGNIZED
- CM_TP_NOT_AVAILABLE_NO_RETRY
- CM_TP_NOT_AVAILABLE_RETRY
- CM_DEALLOCATED_ABEND
- CM_DEALLOCATED_NORMAL
- CM_PROGRAM_ERROR_NO_TRUNC
- CM_PROGRAM_ERROR_PURGING
- CM_PROGRAM_ERROR_TRUNC (basic conversations only)
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_DEALLOCATED_ABEND_SVC (basic conversations only)
- CM_DEALLOCATED_ABEND_TIMER (basic conversations only)
- CM_SVC_ERROR_NO_TRUNC (basic conversations only)
- CM_SVC_ERROR_PURGING (basic conversations only)
- CM_SVC_ERROR_TRUNC (basic conversations only)
- CM_UNSUCCESSFUL
  This value indicates that *receive_type* is set to CM_RECEIVE_IMMEDIATE, but there is no data or status to receive.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR
- The following values are returned only when *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and the conversation is included in a transaction:
  - CM_TAKE_BACKOUT
  - CM_DEALLOCATED_ABEND_BO
  - CM_DEALLOCATED_ABEND_SVC_BO (basic conversations only)
  - CM_DEALLOCATED_ABEND_TIMER_BO (basic conversations only)
  - CM_RESOURCE_FAIL_NO_RETRY_BO
  - CM_RESOURCE_FAILURE_RETRY_BO
  - CM_INCLUDE_PARTNER_REJECT_BO

If a state or parameter error has occurred, *return_code* can have one of the following values:

- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The *receive_type* is set to CM_RECEIVE_AND_WAIT and the conversation is not in **Send**, **Send-Pending**, **Receive** or **Prepared** state.
  - The *receive_type* is set to CM_RECEIVE_IMMEDIATE and the conversation is not in **Receive** or **Prepared** state.
  - The *receive_type* is set to CM_RECEIVE_AND_WAIT; the conversation is basic and in **Send** state; and the program started but did not finish sending a logical record.
  - For a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the conversation's context is in the **Backout-Required** condition. The Receive call is not allowed for this conversation while its context is in this condition.
  - The program has received a *status_received* value of CM_JOIN_TRANSACTION and must issue a tx_begin call to the X/Open TX interface to join the transaction.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *requested_length* specifies a value that exceeds the range permitted by the implementation. The maximum value of the length in each implementation is at least 32767. See Usage Note 13 on page 225 for additional information about determining the maximum buffer size.

***The following return codes apply to full-duplex conversations*.**

If the call is issued in **Send-Receive**, **Receive-Only**, or **Prepared** state, *return_code* can be one of the following:

- CM_OK
- CM_PIP_NOT_SPECIFIED_CORRECTLY
- CM_CONVERSATION_TYPE_MISMATCH
- CM_SECURITY_NOT_VALID
- CM_SYNC_LEVEL_NOT_SUPPORTED_PGM
- CM_SYNC_LEVEL_NOT_SUPPORTED_SYS
- CM_SEND_RCV_MODE_NOT_SUPPORTED
- CM_TPN_NOT_RECOGNIZED
- CM_TP_NOT_AVAILABLE_NO_RETRY
- CM_TP_NOT_AVAILABLE_RETRY
- CM_DEALLOCATED_NORMAL
- CM_DEALLOCATED_ABEND
- CM_DEALLOCATED_ABEND_SVC (basic conversations only)
- CM_DEALLOCATED_ABEND_TIMER (basic conversations only)
- CM_PROGRAM_ERROR_NO_TRUNC
- CM_PROGRAM_ERROR_TRUNC (basic conversations only)
- CM_PROGRAM_ERROR_PURGING (OSI TP CRM only)
- CM_RESOURCE_FAILURE_RETRY
- CM_RESOURCE_FAILURE_NO_RETRY

- CM_SVC_ERROR_NO_TRUNC (basic conversations only)
- CM_SVC_ERROR_TRUNC (basic conversations only)
- CM_OPERATION_NOT_ACCEPTED
- CM_OPERATION_INCOMPLETE
- CM_PRODUCT_SPECIFIC_ERROR
- CM_UNSUCCESSFUL
  This value indicates that *receive_type* is set to CM_RECEIVE_IMMEDIATE, but there is nothing to receive.
- The following values are returned only if *sync_level* is CM_SYNC_POINT_NO_CONFIRM and the state is **Send-Receive** or **Prepared** and the conversation is included in a transaction:
  - CM_TAKE_BACKOUT
  - CM_DEALLOCATED_ABEND_BO
  - CM_DEALLOCATED_ABEND_SCV_BO (basic conversations only)
  - CM_DEALLOCATED_ABEND_TIMER_BO (basic conversations only)
  - CM_RESOURCE_FAIL_NO_RETRY_BO
  - CM_RESOURCE_FAILURE_RETRY_BO
  - CM_CONV_DEALLOC_AFTER_SYNCPT
  - CM_INCLUDE_PARTNER_REJECT_BO

If a state or parameter error has occurred, *return_code* can have one of the following values:

- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The conversation is not in **Send-Receive**, **Prepared**, or **Receive-Only** state.
  - For a conversation with *sync_level* set to CM_SYNC_POINT_NO_CONFIRM, the conversation's context is in the **Backout Required** condition. The Receive call is not allowed for this conversation while its context is in this condition.
  - The local program has received a *status_received* value of CM_JOIN_TRANSACTION and must issue a tx_begin call to the X/Open TX interface to join the transaction.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *requested_length* specifies a value that exceeds the range permitted by the implementation. The maximum value of the length in each implementation is at least 32,767. See Note 13 on page 225 for additional information about determining the maximum buffer size.

State Changes

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

- The conversation enters **Receive** state if a Receive call is issued and all of the following conditions are true:
  - The *receive_type* is set to CM_RECEIVE_AND_WAIT.
  - The conversation is in **Send-Pending** or **Send** state.
  - The *data_received* indicates CM_DATA_RECEIVED, CM_COMPLETE_DATA_RECEIVED, or CM_INCOMPLETE_DATA_RECEIVED.
  - The *status_received* indicates CM_NO_STATUS_RECEIVED.
- The conversation enters **Send** state when *data_received* is set to CM_NO_DATA_RECEIVED and *status_received* is set to CM_SEND_RECEIVED.

- The conversation enters **Send-Pending** state when *data_received* is set to CM_DATA_RECEIVED, or CM_COMPLETE_DATA_RECEIVED, and *status_received* is set to CM_SEND_RECEIVED.
- The conversation enters **Confirm**, **Confirm-Send**, or **Confirm-Deallocate** state when *status_received* is set to, respectively, CM_CONFIRM_RECEIVED, CM_CONFIRM_SEND_RECEIVED, or CM_CONFIRM_DEALLOC_RECEIVED.
- For a conversation with *sync_level* set to either CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the conversation enters **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** state when *status_received* is set to CM_TAKE_COMMIT, CM_TAKE_COMMIT_SEND, or CM_TAKE_COMMIT_DEALLOCATE, respectively.
- For a conversation with *sync_level* set to either CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the conversation enters **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** state when *status_received* is set to CM_TAKE_COMMIT_DATA_OK, CM_TAKE_COMMIT_SEND_DATA_OK, or CM_TAKE_COMMIT_DEALLOC_DATA_OK, respectively.
- No state change occurs when the call is issued in **Receive** state; *data_received* is set to CM_DATA_RECEIVED, CM_COMPLETE_DATA_RECEIVED, or CM_INCOMPLETE_DATA_RECEIVED; and *status_received* indicates CM_NO_STATUS_RECEIVED.
- No state change occurs when the call is issued in **Prepared** state, or if *status_received* indicates CM_JOIN_TRANSACTION.

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

- No state change occurs when the call is issued in **Prepared** or **Receive-Only** state, or if *status_received* indicates CM_JOIN_TRANSACTION.
- The conversation enters **Confirm-Deallocate** state when *status_received* is set to CM_CONFIRM_DEALLOC_RECEIVED.
- For a conversation with *sync_level* set to CM_SYNC_POINT_NO_CONFIRM, the conversation enters **Sync-Point** state when *status_received* is set to CM_TAKE_COMMIT or CM_TAKE_COMMIT_DATA_OK, and it enters **Sync-Point-Deallocate** state when *status_received* is set to CM_TAKE_COMMIT_DEALLOCATE or CM_TAKE_COMMIT_DEALLOC_DATA_OK.

Usage Notes

1. If *receive_type* is set to CM_RECEIVE_AND_WAIT and no information is present when the call is made, CPI Communications waits for information to arrive on the specified conversation before allowing the Receive call to return with the information. If information is already available, the program receives it without waiting.
2. For a half-duplex conversation, if the program issues a Receive call with its end of the conversation in **Send** state with *receive_type* set to CM_RECEIVE_AND_WAIT, the local system will flush its send buffer and send all buffered information to the remote program. The local system will also send a change-of-direction indication. This is a convenient method to change the direction of the conversation, because it leaves the local program's end of the conversation in **Receive** state and tells the remote program that it may now begin sending data. The local system waits for information to arrive.

Note: A Receive call in **Send** or **Send-Pending** state with a *receive_type* set to CM_RECEIVE_AND_WAIT generates an implicit execution of Prepare_To_Receive with *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_FLUSH, followed by a Receive. Refer to “Prepare_To_Receive (CMPTR)” on page 208 for more information.

3. If *receive_type* is set to CM_RECEIVE_IMMEDIATE, a Receive call receives any available information, but does not wait for information to arrive. If information is available, it is returned to the program with an indication of the exact nature of the information received.

   Since data may not be available when a given Receive call is issued, a program that is using concurrent conversations with multiple partners might use a *receive_type* of CM_RECEIVE_IMMEDIATE and periodically check each conversation for data. For more information about multiple, concurrent conversations, see “Multiple Conversations” on page 30.

4. On MVS and VM systems, when a conversation crosses a VTAM network, the Receive call does not return data until an entire logical record arrives at the local system. For basic conversations, this behavior may cause the Receive call to return unexpected results.

   For example, a Receive call with *receive_type* set to CM_RECEIVE_IMMEDIATE will return a return code of CM_UNSUCCESSFUL if the entire logical record has not arrived at the local system, even if enough of the logical record has arrived to satisfy the Receive call's requested length. Similarly, a Receive call with *receive_type* set to CM_RECEIVE_AND_WAIT will wait until the remainder of the logical record is received by the local system, even if enough of the logical record has arrived to satisfy the requested length. For MVS/ESA, the Receive call works properly under these circumstances between LUs controlled by APPC/MVS in the same MVS system image. It also works properly under VM within a TSAF or CS collection.

5. If the *return_code* indicates CM_PROGRAM_STATE_CHECK or CM_PROGRAM_PARAMETER_CHECK, the values of all other parameters on this call have no meaning.

6. A Receive call issued against a mapped conversation can receive only as much of the data record as specified by the *requested_length* parameter. The *data_received* parameter indicates whether the program has received a complete or incomplete data record, as follows:

   - When the program receives a complete data record or the last remaining portion of a data record, the *data_received* parameter is set to CM_COMPLETE_DATA_RECEIVED. The length of the record or portion of the record is less than or equal to the length specified on the *requested_length* parameter.

   - When the program receives a portion of the data record other than the last remaining portion, the *data_received* parameter is set to CM_INCOMPLETE_DATA_RECEIVED. The data record is incomplete for one of the following reasons:

     – *receive_type* is set to CM_RECEIVE_AND_WAIT, and the length of the record is greater than the length specified on the *requested_length* parameter.

     – *receive_type* is set to CM_RECEIVE_IMMEDIATE, and either the length of the record is greater than the length specified on the

requested_length parameter or the last portion of the data record has not arrived from the partner program.

In either case, the amount of data received is equal to the *received_length* specified.

7. When *fill* is set to CM_FILL_LL on a basic conversation, the program intends to receive a logical record, and there are the following possibilities:

   - The program receives a complete logical record or the last remaining portion of a complete record. The length of the record or portion of the record is less than or equal to the length specified on the *requested_length* parameter. The *data_received* parameter is set to CM_COMPLETE_DATA_RECEIVED.
   - The program receives an incomplete logical record for one of the following reasons:
     - The length of the logical record is greater than the length specified on the *requested_length* parameter. In this case, the amount received equals the length specified.
     - Only a portion of the logical record is available (possibly because it has been truncated). The portion is equal to or less than the length specified on the *requested_length* parameter.

     The *data_received* parameter is set to CM_INCOMPLETE_DATA_RECEIVED. The program issues another Receive (or possibly multiple Receive calls) to receive the remainder of the logical record.

   Refer to the Send_Data call for a definition of complete and incomplete logical records.

8. When *fill* is set to CM_FILL_BUFFER on a basic conversation, the program is to receive data independently of its logical-record format. The program receives an amount of data equal to or less than the length specified on the *requested_length* parameter.

   The program can receive less data only under one of the following conditions:

   - *receive_type* is set to CM_RECEIVE_AND_WAIT and the end of the data is received. The end of data occurs when it is followed by either:
     - An indication of a change in the state of the conversation
       - For a half-duplex conversation, a change to **Send**, **Send-Pending**, **Confirm**, **Confirm-Send**, **Confirm-Deallocate**, **Sync-Point**, **Sync-Point-Send**, **Sync-Point-Deallocate**, or **Reset** state
       - For a full-duplex conversation, a change to **Send-Only**, **Confirm-Deallocate**, **Sync-Point**, **Sync-Point-Deallocate**, or **Reset** state
     - An error indication, such as a CM_PROGRAM_ERROR_NO_TRUNC return code.
   - *receive_type* is set to CM_RECEIVE_IMMEDIATE and an amount of data equal to the *requested_length* specified has not arrived from the partner program.

   The program is responsible for tracking the logical-record format of the data.

9. The Receive call made with *requested_length* set to zero has no special significance. The type of information available is indicated by the *return_code*, *data_received*, and *status_received* parameters, as usual. If *receive_type* is

set to CM_RECEIVE_AND_WAIT and no information is available, this call waits for information to arrive. If *receive_type* is set to CM_RECEIVE_IMMEDIATE, it is possible that no information is available.

If data is available, the conversation is basic, and *fill* is set to CM_FILL_LL, the *data_received* parameter indicates CM_INCOMPLETE_DATA_RECEIVED. If data is available, the conversation is basic, and *fill* is set to CM_FILL_BUFFER, the *data_received* parameter indicates CM_DATA_RECEIVED. If data is available and the conversation is mapped, the *data_received* parameter is set to CM_INCOMPLETE_DATA_RECEIVED. In all the above cases, the program receives no data.

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

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

10. The program can receive both data and conversation status on the same call. However, if the remote program truncates a logical record, the local program receives the indication of the truncation on the Receive call issued by the local program after it receives all of the truncated record. The *return_code*, *data_received*, and *status_received* parameters indicate to the program the kind of information the program receives.

11. The program may receive data and conversation status on the same Receive call or on separate Receive calls. The program should be prepared for either case.

12. For a half-duplex conversation, the request-to-send notification is returned to the program in addition to (not in place of) the information indicated by the *return_code*, *data_received*, and *status_received* parameters.

13. A program must not specify a value in the *requested_length* parameter that is greater than the maximum the implementation can support. The maximum may vary from system to system. The program can use the Extract_Maximum_Buffer_Size call to determine the maximum supported by the local system. The program can achieve portability across different systems by using one of the following methods:

    a. Never using a *requested_length* value greater than 32767.

    b. Using the Extract_Maximum_Buffer_Size call to determine the maximum buffer size supported by the system and never setting *requested_length* greater than that maximum buffer size.

    The program should also be aware that the CM_INCOMPLETE_DATA_RECEIVED value of the *data_received* parameter may be returned when the maximum buffer size differs across the systems.

14. When the Receive call is processed in non-blocking mode and *receive_type* is set to CM_RECEIVE_IMMEDIATE, the call completes immediately. If information is not available, *return_code* is set to CM_UNSUCCESSFUL.

15. When the local program has requested confirmation of the Allocate call and the first call made by the recipient program is Request_To_Send or the recipient

program has issued a Send_Expedited_Data call, the CM_ALLOCATE_CONFIRMED value of the *control_information_received* parameter will be returned first, and one of the following values will be returned at the next opportunity:

- CM_RTS_RCVD_AND_EXP_DATA_AVAIL
- CM_REQ_TO_SEND_RECEIVED
- CM_EXPEDITED_DATA_AVAILABLE

16. The Receive call may be issued following a successful Prepare call, without a state transition to **Receive** state in case of a half-duplex conversation, for either of these reasons:

    - To receive data when CM_PREPARE_DATA_PERMITTED is selected and the Prepare Call is issued
    - To receive a new CM_PREPARE_OK value in *status_received*.

17. For a full-duplex conversation, if *receive_type* is set to CM_RECEIVE_AND_WAIT and the conversation startup request has not been sent to the partner, then the Receive call will flush the conversation startup request to the partner.

18. For a full-duplex conversation, if the return code CM_PROGRAM_ERROR_PURGING is received, it indicates that the conversation is allocated using an OSI TP CRM and that data may have been purged. The application has to ensure that the two partners are coordinated.

19. For a full-duplex conversation, when CM_DEALLOCATED_ABEND or CM_DEALLOCATED_ABEND_BO is received, further information on the cause of the deallocation may be obtained by issuing the Extract_Secondary_Information call.

20. When *control_information_received* indicates that expedited data is available to be received, subsequent calls with this parameter will continue to indicate that expedited data is available until the expedited data has been received by the program.

Related Information

"Conversation Types" on page 19 and "Set_Fill (CMSF)" on page 310 further discuss the use of basic conversations.

Most of the example program flows in Chapter 3, "Program-to-Program Communication Example Flows" show programs using the Receive call.

"Extract_Maximum_Buffer_Size (CMEMBS)" on page 173 further discusses determining the maximum buffer size supported by the system.

"Receive_Mapped_Data (CMRCVM)" on page 231 discusses how a program receives partner mapped data.

"Request_To_Send (CMRTS)" on page 246 discusses how a program can place its end of the conversation into **Receive** state.

"Send_Data (CMSEND)" on page 249 provides more information on complete and incomplete logical records and data records.

"Send_Mapped_Data (CMSNDM)" on page 271 dicusses how a program sends mapped partner data.

"Set_Receive_Type (CMSRT)" on page 344 discusses the *receive_type* characteristic and its various values.

Receive_Expedited_Data (CMRCVX)

LU 6.2

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
					X*			

A program uses the Receive_Expedited_Data (CMRCVX) call to receive expedited data sent by its partner.

This call has meaning only when an LU 6.2 CRM is used for the conversation.

X* In OS/2, this call is supported in Communications Server.

Format

CALL CMRCVX(*conversation_ID*,
buffer,
requested_length,
received_length,
control_information_received,
expedited_receive_type,
return_code)

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

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

requested_length *(input)*
Specifies the maximum amount of data the program is to receive. This length can range from 0 to 86 bytes.

received_length *(output)*
When CM_OK is returned to the program, this parameter specifies the amount of data received, which is less than or equal to the buffer size specified in *requested_length*. When CM_BUFFER_TOO_SMALL is returned to the program, this parameter indicates the size of the data that is available to be received but has not been received.

control_information_received *(output)*
Specifies the variable containing an indication of whether or not control information has been received.

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

- CM_NO_CONTROL_INFO_RECEIVED
  Indicates that no control information was received.
- CM_REQ_TO_SEND_RECEIVED (half-duplex conversations only)
  The local program received a request-to-send notification from the remote program. The remote program issued Request_To_Send, requesting the

local program's end of the conversation to enter **Receive** state, which would place the remote program's end of the conversation in **Send** state. See "Request_To_Send (CMRTS)" on page 246 for further discussion of the local program's possible responses.

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

Notes:

1. If *return_code* is set to CM_PROGRAM_PARAMETER_CHECK or CM_PROGRAM_STATE_CHECK, the value contained in *control_information_received* has no meaning.

2. When more than one piece of control information is available to be returned to the program, it will be returned in the following order:

   - CM_RTS_RCVD_AND_EXP_DATA_AVAIL
   - CM_REQ_TO_SEND_RECEIVED
   - CM_EXPEDITED_DATA_AVAILABLE
   - CM_NO_CONTROL_INFO_RECEIVED

expedited_receive_type *(input)*
Specifies whether control should be returned to the program immediately or after there is expedited data available to receive.

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

- CM_RECEIVE_AND_WAIT
  The Receive_Expedited_Data call is to wait for expedited data to arrive on the specified conversation. If expedited data is already available, the program receives it without waiting.
- CM_RECEIVE_IMMEDIATE
  The Receive_Expedited_Data call is to receive any expedited data that is available from the specified conversation, but is not to wait for expedited data to arrive.

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_PROGRAM_PARAMETER_CHECK
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *requested_length* specifies a value less than 0 or greater than 86.
  - The conversation is not using an LU 6.2 CRM.
  - The *expedited_receive_type* specifies an undefined value.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is in **Initialize** or **Initialize-Incoming** state and is not allowed to send expedited data.
- CM_CONVERSATION_ENDING
  This value indicates that the conversation is ending due to a normal deallocation, an allocation error, a Cancel_Conversation call, a Deallocate

call with *deallocate_type* set to CM_DEALLOCATE_ABEND, or a conversation failure. Hence, no expedited data is received.

- CM_EXP_DATA_NOT_SUPPORTED
  This value indicates that the remote system does not support expedited data.
- CM_BUFFER_TOO_SMALL
  This value indicates that the value specified for the *requested_length* parameter is less than the amount of expedited data to be received. Therefore, no expedited data has been received.
- CM_OPERATION_NOT_ACCEPTED
- CM_UNSUCCESSFUL
  This value indicates that the *expedited_receive_type* parameter was set to CM_RECEIVE_IMMEDIATE and there was no expedited data available to receive.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. When expedited data is received by the CRM from the partner program, it is indicated to the local program on the next call it issues that returns the *control_information_received* parameter. When the program uses multiple threads or queue-level non-blocking, more than one call with the *control_information_received* parameter may be executed simultaneously. The availability of expedited data will continue to be indicated until the expedited data is received by the program. However, if a request-to-send or an allocate-confirm notification has been received, this notification is given to the program in only one call that has the *control_information_received* parameter.

2. If the program issues Receive_Expedited_Data with *requested_length* set to 0 and there is data available to be received, CM_BUFFER_TOO_SMALL is returned.

3. If the program issues Receive_Expedited_Data with *requested_length* set to 0 and *expedited_receive_type* set to CM_RECEIVE_AND_WAIT, and there is no data available to be received, the call does not complete until expedited data is available to be received. CM_BUFFER_TOO_SMALL is then returned.

Related Information

"Send_Expedited_Data (CMSNDX)" on page 268 describes the Send_Expedited_Data call.

Receive_Mapped_Data (CMRCVM)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32

A program issues the Receive_Mapped_Data (CMRCVM) call to receive mapped data sent by its partner.

Before issuing the Receive_Mapped_Data call, a program has the option of issuing the CMSRT-Set_Receive_Type call.

Format

```
CALL CMRCVM(conversation_ID,
            map_name,
            map_name_length,
            buffer,
            requested_length,
            data_received,
            status_received,
            control_information_received,
            return_code)
```

Parameters

conversation_ID *(input)*
Specifies the variable containing the conversation identifier assigned to the conversation.

map_name *(output)*
Specifies the variable containing the mapping function used to decode the data record. The length of the variable must be at least 64 bytes.

map_name_length *(output)*
Specifies the variable containing the length of the returned map_name parameter.

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

Note: *Buffer* contains data only if *return_code* is set to CM_OK or CM_DEALLOCATE_NORMAL and *data_received* is not set to CM_NO_DATA_RECEIVED. If the *return_code* is set to CM_MAP_ROUTINE_ERROR or CM_UNKNOWN_MAP_RECEIVED the buffer contains the unprocessed user data.

requested_length *(input)*
Specifies the maximum amount of data the program is to receive. Valid *requested_length* values range from 0 to the maximum buffer size supported by the system and the map name. See usage note 10 on page 241 for additional information about determining the maximum buffer size.

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

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

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

- CM_NO_DATA_RECEIVED
  No data is received by the program. Status may be received if the *return_code* is set to CM_OK
- CM_COMPLETE_DATA_RECEIVED
  A complete data record or the last remaining portion of the record is received.
- CM_INCOMPLETE_DATA_RECEIVED

Less than a complete data record was received.

received_length *(output)*
Specifies the variable containing the amount of data the program received, up to the maximum. If the program does not receive data on this call, the value contained in *received_length* has no meaning.

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

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

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

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

- CM_NO_STATUS_RECEIVED
  No conversation status is received by the program; data may be received.
- CM_SEND_RECEIVED (half-duplex conversations only)
  The remote program's end of the conversation in **Send-Pending** state (if the program also received data on this call) or **Send** state (if the program did not receive data on this call). The local program which issued the Receive call can now send data.
- CM_CONFIRM_RECEIVED (half-duplex conversations only)
  The remote program has sent a confirmation request, requesting the local program to respond by issuing a Confirmed call. The local program must respond by issuing Confirmed, Send_Error, or Deallocate with *deallocate_type* set to CM_DEALLOCATE_ABEND.
- CM_CONFIRM_SEND_RECEIVED (half-duplex conversations only)
  The remote program's end of the conversation entered **RECEIVE** state with conformation requested. The local program must respond by issuing Confirmed, Send_Error, or Deallocate with *deallocate_type* set to CM_DEALLOCATE_ABEND. Upon issuing a successful Confirmed call, the local program which issued the Receive call can now send data.
- CM_CONFIRM_DEALLOC_RECEIVED
  The remote program has deallocated the conversation with conformation requested. The local program must respond by issuing Confirmed, Send_Error, or Deallocate with *deallocate_type* set to CM_DEALLOCATE_ABEND. Upon

issuing a successful Confirmed call, the local program which issued the Receive call is deallocated. This means it is placed in **RESET** state.

For a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and with the conversation include in a transaction, the *status_received* variable can also be set to one of the following values:

- CM_TAKE_COMMIT
  The remote program has issued a resource recovery commit call. The local program should issue a commit call in order to commit all protected program should issue a commit call in order to commit all protected resources throughout the transaction. When appropriate, the local program may respond by issuing a call other than commit, such as Send_error (for half-duplex conversations only) or a resource recovery backout call.
- CM_TAKE_COMMIT_SEND (half-duplex conversations only)
  The remote program issued a Prepare_to_Receive call with *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync level* set to CM_SYNC_POINT and then issued a resource recovery commit call. The local program should issue a commit call in order to commit all protected resources throughout the transaction. When appropriate, the local program may respond by issuing a call other than commit, such as Send_Error, or a resource recovery backout call. If a successful commit call is issued, the local program can then send data.
- CM_TAKE_COMMIT_DEALLOCATE
  The remote program has deallocated the conversation with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_COMMIT and then issued a resource recovery commit call. The local program should issue a commit call in order to commit all protected resources throughout the transaction. The local program may respond by issuing a call other than commit, such as Send_Error, or a resource recovery backout call. If a successful commit call is issued, the local program is then deallocated. This means it is placed in **RESET** state.
- CM_PREPARE_OK
  By issuing a Prepare (CMPREP) call, the local program requested that the remote program prepare its resources for commitment, and the remote program has done so by issuing a commit call. The subtree is now ready to commit its resources.

For a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and allocated using an OSI TP CRM, the *status_received* variable can also be set to one of the values if the conversation is included in a transaction:

- CM_JOIN_TRANSACTION
  The remote program requested that the local program join into its current transaction. The local program should issue an *tx_begin* call to the X/OPEN TX interface to join the superior's transaction as soon as any local work that is not to be included in the remote program's transaction has been completed.
- CM_TAKE_COMMIT_DATA_OK
  The remote program issued a Prepare call. For the exact conditions for receipt of this status_received value, see Table 17 on page 342 and Table 59 on page 642 The local program should issue a commit call in order to commit all protected resources throughout the transaction. The program is allowed to send data before issuing the commit call. When appropriate, the local program

may respond by issuing a call other than commit, such as Send_Error for half-duplex conversations only, or a resource recovery backout call.

- CM_TAKE_COMMIT_SEND_DATA_OK(half-duplex conversations only)
  The remote program issued a Prepare call. For the exact conditions for receipt of this status_received value, see Table 17 on page 342 and Table 59 on page 642 The local program should issue a commit call in order to commit all protected resources throughout the transaction. The program is allowed to send data before issuing the commit call. When appropriate, the local program may respond by issuing a call other than commit, such as Send_Error for half-duplex conversations only, or a resource recovery backout call. If a successful commit call is issued, the local program can then send data.
- CM_TAKE_COMMIT_DEALLOC_DATA_OK
  The remote program issued a Prepare call. For the exact conditions for receipt of this status_received value, see Table 17 on page 342 and Table 59 on page 642 The local program should issue a commit call in order to commit all protected resources throughout the transaction. The program is allowed to send data before issuing the commit call. When appropriate, the local program may respond by issuing a call other than commit, such as Send_Error for half-duplex conversations only, or a resource recovery backout call. If a successful commit call is issued, the local program is then deallocated. This means that it is placed in **RESET** state.

control_information_received (output)
Specifies the variable containing an indication of whether or not control information has been received. The *control_infomration_received* variable can have one of the following values:

- CM_NO_CONTROL_INFO_RECEIVED
  Indicates that no control information was received.
- CM_REQ_TO_SEND_RECEIVED (half-duplex conversations only)
  The local program received a request-to-send notification from the remote program. The remote program issued Request_To_Send, requesting the local program's end of conversation to enter **Receive** state, which would place the remote program's end of conversation in **Send** state. See "Request_To_Send (CMRTS)" on page 246 for further discussion of the local program's possible responses.
- CM_ALLOCATE_CONFIRM (OSI TP CRM only)
  The local program received confirmation of the remote program's acceptance of the conversation.
- CM_ALLOCATE_CONFIRM_WITH_DATA (OSI TP CRM only)
  The local program received confirmation of the remote program's acceptance of the conversation. The local program may now issue an Extract_Mapped_Initialization_Data (CMEMID) call to receive the initialization data.
- CM_ALLOCATE_REJECT_WITH_DATA (OST TP CRM only)
  The remote program rejected the conversation. The local program may now issue an Extract_Mapped_Initialization_call to receive the initialization data.

  This value will be returned with a return code of CM_OK. The program will receive a CM_DEALLOCATE_ABEND return code on a later call on the conversation.
- CM_EXPEDITED_DATA_AVAILABLE (LU 6.2 CRM only)
  Expedited data is available to be received.
- CM_EXPEDITED_MAPPED_DATA_AVAILABLE (LU 6.2 CRM only)
  Mapped expedited data is available to be received.

- CM_RTS_RCVD_AND_EXP_MAP_DATA_AVAIL (half-duplex conversations and LU 6.2 CRM only)
  The local program received a request-to-send notification from the remote program and mapped expedited date is available for receiving.

Notes:

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

return_code (output)

Specifies the result of call execution The return_code that can be returned depends on the state and characteristics of he conversation at the time this call is issued.

The following return codes apply to half-duplex conversations:

If *receive_type* is set to CM_RECEIVE_AND_WAIT and this call is issued in **Send** state, the return_code can have one of the following values:

- CM_OK
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_CONVERSATION_TYPE_MISMATCH
- CM_PIP_NOT_SPECIFIED_CORRECTLY
- CM_SECURITY_NOT_VALID
- CM_SYNC_LVL_NOT_SUPPORTED_PGM
- CM_SYNC_LVL_NOT_SUPPORTED_SYS
- CM_TPN_NOT_RECOGNIZED
- CM_TPN_NOT_AVAILABLE_NO_RETRY
- CM_TPN_NOT_AVAILABLE_RETRY
- CM_DEALLOCATE_ABEND
- CM_DEALLOCATE_NORMAL
- CM_PROGRAM_ERROR_NO_TRUNC
- CM_PROGRAM_ERROR_NO_PURGING
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_OPERATION_NOT_ACCEPTED
- CM_UNKNOWN_MAP_NAME_RECEIVED

The received data requires a map routine that the local map function does not support. The buffer contains the unprocessed user data.

- CM_MAP_ROUTINE_ERROR
  The map routine encountered a problem with the received data. The buffer contains the unprocessed user data.
- CM_PRODUCT_SPECIFIC_ERROR
- CM_PROGRAM_STATE_CHECK
- CM_PROGRAM_STATE_CHECK

  The following values are returned only when the *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and the conversation is included in the transaction:
  - CM_TAKE_BACKOUT
  - CM_DEALLOCATED_ABEND_BO
  - CM_RESOURCE_FAIL_NO_RETRY_BO
  - CM_RESOURCE_FAILURE_RETRY_BO

If *receive_type* is set to CM_RECEIVE_AND_WAIT and this call is issued in **Send-Pending** state, *return_code* can have one of the following values:

- CM_OK
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_SYNC_LVL_NOT_SUPPORTED_SYS
- CM_DEALLOCATED_ABEND
- CM_DEALLOCATED_NORMAL
- CM_PROGRAM_ERROR_NO_TRUNC
- CM_PROGRAM_ERROR_PURGING
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_UNKNOWN_MAP_NAME_RECEIVED
  The received data requires a map routine that the local map function does not support. The buffer contains the unprocessed user data.
- CM_MAP_ROUTINE_ERROR
  The map routine encountered a problem with the received data. The buffer contains the unprocessed user data.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

The following values are returned only when the *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and the conversation is included in a transaction:

- CM_TAKE_BACKOUT
- CM_DEALLOCATED_ABEND_BO
- CM_RESOURCE_FAIL_NO_RETRY_BO
- CM_RESOURCE_FAILURE_RETRY_BO

If *receive_type* is set to CM_RECEIVE_AND_WAIT or CM_RECEIVE_IMMEDIATE and this call is issued in **Receive** state, the return_code can have one of the following values:

- CM_OK
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
  This value is only received when *receive_type* is set to CM_RECEIVE_AND_WAIT.
- CM_CONVERSATION_TYPE_MISMATCH

- CM_PIP_NOT_SPECIFIED_CORRECTLY
- CM_SECURITY_NOT_VALID
- CM_SYNC_LVL_NOT_SUPPORTED_PGM
- CM_SYNC_LVL_NOT_SUPPORTED_SYS
- CM_TPN_NOT_RECOGNIZED
- CM_TPN_NOT_AVAILABLE_NO_RETRY
- CM_TPN_NOT_AVAILABLE_RETRY
- CM_DEALLOCATE_ABEND
- CM_DEALLOCATE_NORMAL
- CM_PROGRAM_ERROR_NO_TRUNC
- CM_PROGRAM_ERROR_PURGING
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_UNSUCCESSFUL
- CM_UNKNOWN_MAP_NAME_RECEIVED
  The received data requires a map routine the local map function does not support. The buffer contains the unprocessed user data.
- CM_MAP_ROUTINE_ERROR
  The map routine encountered a problem with the received data. The buffer contains the unprocessed user data.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

The following values are returned only when the *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFORM and the conversation is included in a transaction:

- CM_TAKE_BACKOUT
- CM_DEALLOCATE_ABEND_BO
- CM_RESOURCE_FAIL_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY

If the state or parameter error has occurred, the *return_code* can have one of the following values:

- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
- The *receive_type* is set to CM_RECEIVE_AND_WAIT and the conversation is not set to CM_RECEIVE_AND_WAIT and the conversation is not in **Send, Send-Pending, Receive or Prepare** state.
- The *receive_type* is set to CM_RECEIVE_IMMEDIATE and the conversation is not set to CM_RECEIVE_AND_WAIT and the conversation is not in **Receive or Prepare** state.
- For a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the conversation's context is in the **Backout-Required** condition. The Receive_Mapped_Data call is not allowed for this conversation while its context is in this condition.
- The program has a *status_received* value of CM_JOIN_TRANSACTION and must issue a *tx_begin* call to the X/OPEN TX interface to join the transaction.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
- The *conversation_ID* specifies an unassigned conversation identifier.
- The requested_length specifies a value that exceeds the range permitted by implementation. The maximum value of the length in each implementation is at

least 32,767. See usage note 10 on page 241 for additional information about determining the maximum buffer size.
- The conversation_type characteristic is set to CM_BASIC_CONVERSATION

The following return codes apply to full_duplex conversations:

If the call is issued in **Send-Receive** or **Receive-Only** state and either the *sync_level* is CM_NONE or the *sync_level* is CM_SYNC_POINT_NO_CONFIRM and the conversation is not currently included in a transaction, the return_code can have one of the following values:

- CM_CONVERSATION_TYPE_MISMATCH
- CM_PIP_NOT_SPECIFIED_CORRECTLY
- CM_SECURITY_NOT_VALID
- CM_SYNC_LVL_NOT_SUPPORTED_PGM
- CM_SYNC_LVL_NOT_SUPPORTED_SYS
- CM_SEND_RCV_MODE_NOT_SUPPORTED
- CM_TPN_NOT_RECOGNIZED
- CM_TPN_NOT_AVAILABLE_NO_RETRY
- CM_TPN_NOT_AVAILABLE_RETRY
- CM_DEALLOCATE_ABEND
- CM_DEALLOCATE_NORMAL
- CM_PROGRAM_ERROR_NO_TRUNC (half-duplex conversations only)
- CM_PROGRAM_ERROR_PURGING (OSI TP CRM only)
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_OPERATION_NOT_ACCEPTED
- CM_UNKNOWN_MAP_NAME_RECEIVED
  The received data requires a map routine the local map function does not support. The buffer contains the unprocessed user data.
- CM_MAP_ROUTINE_ERROR
  The map rountine encountered a problem with the received data. The buffer contains the unprocessed user data.
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_PRODUCT_SPECIFIC_ERROR
- CM_PROGRAM_STATE_CHECK

The following values are returned only when the *sync_level* is set to CM_SYNC_POINT_NO_CONFORM and the state is **Send-Receive** or **Prepared** and the conversation is included in a transaction:

- CM_TAKE_BACKOUT
- CM_DEALLOCATE_ABEND_BO
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_CONV_DEALLOC_AFTER_SYNCPT
- CM_INCLUDE_PARTNER_REJECT_BO
- CM_PRODUCT_SPECIFIC_ERROR_

If the state or parameter error has occurred, the *return_code* can have one of the following values:

CM_PROGRAM_STATE_CHECK
This value indicates one of the following:
The conversation is not in **Send-Receive, Prepared** or **Receive-Only** state.
For a conversation with *sync_level* set to CM_SYNC_POINT_NO_CONFIRM, the conversation's context is in the **Backout-Required** condition. The Receive_Mapped_Data call is not allowed for this conversation while its context is in this condition.
CM_PROGRAM_PARAMETER_CHECK

This value indicates one of the following:

The *conversation_ID* specifies an unassigned conversation identifier.
The requested_length specifies a value that exceeds the range permitted by implementation. The maximum value of the length in each implementation is at least 32,767. See usage note 10 on page 241 for additional information about determining the maximum buffer size.
The conversation_type characteristic is set to CM_BASIC_CONVERSATION

State Changes

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

The conversation enters Receive state if a Receive call is issued and all of the following conditions are true:
- The receive type is set to CM_RECEIVE_AND_WAIT.
- The conversation is in **Send-Pending** or **Send** state.
- The *data_received* indicates CM_DATA_RECEIVED, CM_COMPLETE_DATA_RECEIVED or CM_INCOMPLETE_DATA_RECEIVED
- The *status_received* indicates CM_NO_STATUS_RECEIVED.

- The conversation enters **Send** state when *data_received* is set to CM_NO_DATA_RECEIVED and *status_received* is set to CM_SEND_RECEIVED.
- The conversation enters **Send-Pending** state when *data_received* is set to CM_DATA_RECEIVED, or CM_COMPLETE_DATA_RECEIVED, and *status_received* is set to CM_SEND_RECEIVED.
- The conversation enters **Confirm**, **Confirm-send**, or **Confirm-Deallocate** state when *status_received* is set to CM_CONFIRM_RECEIVED, CM_CONFIRM_SEND_RECEIVED, or CM_CONFIRM_DEALLOC_RECEIVED, respectively.
- For a conversation with *sync_level* set to either CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the conversation enters **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** state when *status_received* is set to CM_TAKE_COMMIT, CM_TAKE_COMMIT_SEND, or CM_TAKE_COMMIT_DEALLOCATE, respectively.
- For a conversation with *sync_level* set to either CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the conversation enters **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** state when *status_received* is set to CM_TAKE_COMMIT_DATA_OK, CM_TAKE_COMMIT_SEND_DATA_OK, or CM_TAKE_COMMIT_DEALLOC_DATA_OK, respectively.
- No state change occurs when the call is issued in **Receive** state. *data_received* is set to CM_DATA_RECEIVED, CM_COMPLETE_DATA_RECEIVED, or CM_INCOMPLETE_DATA_RECEIVED and *status_received* indicates CM_NO_STATUS_RECEIVED

- No state change occurs when the call is issued in **Prepared** state, or if *status_received* indicates CM_JOIN_TRANSACTION.

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

- No state change occurs when the call is issued in **Prepared** or **Receive-Only** state, or if *status_received* indicates CM_JOIN_TRANSACTION
- The conversation enters **Confirm-Deallocate** state when *status_rejected* is set to CM_CONFIRM_DEALLOC_RECEIVED.
- For a conversation with *sync_level* set to CM_SYNC_POINT_NO_CONFIRM, the conversation enters **Sync-Point** state when *status_received* is set to CM_TAKE_COMMIT or CM_TAKE_COMMIT_DATA_OK, and it enters **Sync-Point-Deallocate** state when *status_received* is set to CM_TAKE_COMMIT_DEALLOCATE, or CM_TAKE_COMMIT_DEALLOC_DATA_OK.

Usage Notes

1. If *receive_type* is set to CM_RECEIVE_AND_WAIT and no information is present when the call is made, CPI-C waits for information to arrive on the specified conversation before allowing the Receive call to return with the information. If the information is already available, the program receives it without waiting.

2. For a half-duplex conversation, if the program issues a Receive call with its end of the conversation in **Send** state with *receive_type* set to CM_RECEIVE_AND_WAIT, the local system will flush its send buffer and send all buffered information to the remote program. The local system will also send a change-of-direction indication. This is convenient method to change the direction of the conversation because it leaves the local program's end of the conversation in Receive state and tells the remote program that it may now begin sending data. The local system waits for information to arrive.

   Note: A Receive call in **Send** or **Send-Pending** state with a *receive_type* set to CM_RECEIVE_AND_WAIT generates an implicit execution of Prepare_To_Receive with *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_FLUSH followed by a Receive. See “Prepare_To_Receive (CMPTR)” on page 208 for more information.

3. If *receive_type* is set to CM_RECEIVE_IMMEDIATE, a Receive call receives any available information, but does not wait for information to arrive. If information is available, it is returned to the program with an indication of the exact nature of the information received.

   Since data may not be available when a given Receive call is issued, a program that is using concurrent conversations with multiple partners might use a *receive_type* of CM_RECEIVE_IMMEDIATE and periodically check each conversation for data. For more information about multiple, concurrent conversations see “Multiple Conversations” on page 30

4. If the *return_code* indicates CM_PROGRAM_STATE_CHECK or CM_PROGRAM_PARAMETER_CHECK, the values of all other parameters on the call have no meaning.

5. A Receive call is issued against a mapped conversation can receive only as much of the data record as specified by the *requested_length* parameter. The *data_received* parameter indicates whether the program has received a complete or incomplete data record as follows:

- When the program receives a compete data record or the last remaining portion of a data record, the *data_received* is set to CM_COMPLETE_DATA_RECEIVED The length of the record or portion of the record is less than or equal to the length specified on the *requested_length* parameter.
- When the program receives a portion of the data record other than the last remaining portion, the *data_received* parameter is set to CM_INCOMPLETE_DATA_RECEIVED. The data record is incomplete for one of the following reasons:
  - *receive_type* is set to CM_RECEIVE_AND_WAIT and the length of the record is greater than the length specified on the *requested_length* parameter.
  - *receive_type* is set to CM_RECEIVE_IMMEDIATE and either the length of the record is greater than the length specified on the *requested_length* parameter or the last portion of the data record has not arrived from the partner program.
    In either case, the amount of data received is equal to the *received_length* specified.

6. The Receive call made with *requested_length* set to zero has no special significance. The type of information available is indicated by the *return_code*, *data_received*. and *status_received* parameters as usual. If *receive_type* is set to CM_RECEIVE_AND_WAIT and no information is available, this call waits for information to arrive. If *receive_type* is set to CM_RECEIVE_IMMEDIATE, it is possible that no information is available. If data is available, the *data_received* parameter is set to CM_INCOMPLETE_DATA_RECEIVED and the program receives no data.

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

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

7. The program can receive both data and conversation status on the same call. However, if the remote program truncates a logical record, the local program receives the indication of the truncation on the Receive call issued by the local program after it receives all of the truncated record. The *return_code*, *data_received*, and *status_received* parameters indicate to the program the kind of information the program receives.
8. The program may receive data and conversation status on the same Receive call or on separate Receive calls. The program should be prepared for either case.
9. For a half-duplex conversation, the request-to-send notification is returned to the program in addition to (not in place of) the information indicated by the *return_code_data_received* and *status_received* parameters.
10. A program must not specify a value in the *requested_length* parameter that is greater than the maximum the implementation can support. The maximum may

vary from system to system. The maximum also depends on the *map_name* parameter.

Every CPI-C product must be able to receive data buffers with a length of 32,767 bytes. Users who write map routines should describe the length of the data buffer after decoding, if the encoded buffer has a length of 32,767 bytes.

The application program should use a *requested_length* value not smaller than this value. The program should also be aware that CM_INCOMPLETE_DATA_RECEIVED value of the *data_received* parameter may be returned when the maximum buffer size differs across systems.

11. When the Receive call is processed in non-blocking mode and *receive_type* is set to CM_RECEIVE_IMMEDIATE, the call completes immediately. If information is not available, *return_code* is set to CM_UNSUCCESSFUL.

12. When the local program has requested confirmation of the Allocate call and the first call made by the recipient program is Request_To_Send or the recipient program has issued a Send_Expedited_Data call, the CM_ALLOCATE_CONFIRMED value of the *control_information_received* parameter will be returned first and one of the following values will be returned at the next opportunity:

    - CM_RTS_RCVD_AND_EXP_DATA_AVAIL
    - CM_REQ_TO_SEND_RECEIVED
    - CM_EXPEDITED_DATA_AVAILABLE
    - CM_EXPEDITED_MAPPED_DATA_AVAILABLE

13. The Receive call may be issued following a successful Prepare call, without a state transition to **Receive** state in case of a half-duplex conversation, for either of these reasons:

    - To receive data when CM_PREPARE_DATA_PERMITTED is selected and the Prepare Call is issued
    - To receive a new CM_PREPARE_OK value in *status_received*.

14. For a full-duplex conversation, if *receive_type* is set to CM_RECEIVE_AND_WAIT and the conversation startup request has been sent to the partner, then the Receive call will flush the conversation startup request to the partner.

15. For a full-duplex conversation, if the return code CM_PROGRAM_ERROR_PURGING is received, it indicates that the conversation is allocated using an OSI TP CRM and that data may have been purged. The application has to ensure that the two partners are coordinated.

16. For a full-duplex conversation, when CM_DEALLOCATED_ABEND or CM_DEALLOCATED_ABEND_BO is received, further information on the cause of the deallocation may be obtained by issuing the Extract_Secondary_Information call.

17. When *control_information_received* indicated that expedited data is available to be received, subsequent calls with this parameter will continue to indicate that expedited data is available until the expedited data has been received by the program.

18. The *received_length* is local information from the map routine. The *received_length* value is calculated by the map routine after all transformations are completed.

19. If a basic conversation data record is received the map routine treats it as an error.

Related Information

Chapter 3, "Program-to-Program Communication Example Flows" on page 67 show programs using the Receive call.

"Request_To_Send (CMRTS)" on page 246 discusses how a program can place its end of the conversation into **Receive** state.

"Send_Mapped_Data (CMSNDM)" on page 271 provides more information on complete and incomplete logical records and data records.

"Set_Prepare_To_Receive_Type (CMSPTR)" on page 331 discusses the *receive_type* characteristic and its various values.

Release_Local_TP_Name (CMRLTP)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X*					X*			X

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

X* In AIX, this call is supported in Version 3 Release 1 or later. OS/2, this call is supported in Communications Server.

Format

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

Parameters

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

Note: Refer to "SNA Service Transaction Programs" on page 727 for special handling of SNA Service Transaction Program names.

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

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  – The *TP_name* specifies a name that is not associated with this program.
  – The *TP_name_length* specifies a value less than 1 or greater than 64.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

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

Related Information

"Specify_Local_TP_Name (CMSLTP)" on page 361 describes how local names are associated with a program.

Request_To_Send (CMRTS)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

The local program uses the Request_To_Send (CMRTS) call to notify the remote program that the local program would like to enter **Send** state for a given conversation.

Note: The Request_To_Send call has meaning only on a half-duplex conversation.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

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

- CM_OK
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  – The *conversation_ID* specifies an unassigned conversation identifier.
  – The *send_receive_mode* is CM_FULL_DUPLEX.
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  – The conversation is not in **Receive**, **Send**, **Send-Pending**, **Confirm**, **Confirm-Send**, **Confirm-Deallocate**, **Sync-Point**, **Sync-Point-Send**, **Sync-Point-Deallocate** or **Prepared** state.
  – For a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the conversation's context is in the **Backout-Required** condition. The Request_To_Send call is not allowed for this conversation while its context is in this condition.
  – For a conversation using an OSI TP CRM, the Request_To_Send call is not allowed from **Send** or **Prepared** state.
  – The program has received a *status_received* value of CM_JOIN_TRANSACTION and must issue a tx_begin call to the X/Open TX interface to join the transaction.
- CM_CONVERSATION_ENDING

This return code indicates that the local system is ending the conversation or notification has been received from the remote system that it is ending the conversation.

- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. The remote program is informed of the arrival of a request-to-send notification by means of the *control_information_received* parameter. The *control_information_received* parameter set to CM_REQ_TO_SEND_RECEIVED or CM_RTS_RCVD_AND_EXP_DATA_AVAIL is a request for the remote program's end of the conversation to enter **Receive** state in order to place the partner program's end of the conversation (the program that issued the Request_To_Send) in **Send** state.

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

   - The Receive call with *receive_type* set to CM_RECEIVE_AND_WAIT
   - The Prepare_To_Receive call with *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_FLUSH, CM_PREP_TO_RECEIVE_CONFIRM, or CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_CONFIRM or CM_NONE, or CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, but with the conversation not currently included in a transaction.
   - The Send_Data call with *send_type* set to CM_SEND_AND_PREP_TO_RECEIVE and *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_FLUSH, CM_PREP_TO_RECEIVE_CONFIRM, or CM_PREP_TO_RECEIVE_SYNC_LEVEL, and *sync_level* set to CM_CONFIRM or CM_NONE, or CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, but with the conversation not currently included in a transaction.
   - The Prepare_To_Receive call with *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, followed by a successful commit, Confirm, or Flush call.
   - The Send_Data call with *send_type* set to CM_SEND_AND_PREP_TO_RECEIVE, *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, and *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, followed by a successful commit, Confirm, or Flush call.

   After a remote program issues one of these calls, the local program's end of the conversation is placed into a corresponding **Send**, **Send-Pending**, **Confirm-Send**, or **Sync-Point-Send** state when the local program issues a Receive call. See the *status_received* parameter for the Receive call on page 213 for information about why the state changes from **Receive** to **Send**.

2. The CM_REQ_TO_SEND_RECEIVED value is normally returned to the remote program in the *control_information_received* parameter when the remote program's end of the conversation is in **Send** state (on a Send_Data,

Send_Error, Confirm, or Test_Request_To_Send_Received call). However, the value can also be returned on a Receive call.

3. When the remote system receives the request-to-send notification, it retains the notification until the remote program issues a call with the *control_information_received* parameter. The remote system will retain only one request-to-send notification at a time (per conversation). Additional notifications are discarded until the retained notification is indicated to the remote program. Therefore, a local program may issue the Request_To_Send call more times than are indicated to the remote program.

Related Information

"Example 4: The Receiving Program Changes the Data Flow Direction" on page 75 shows an example program flow using the Request_To_Send call.

"Receive (CMRCV)" on page 213 provides additional information on the *status_received* and *control_information_received* parameters.

"Receive_Mapped_Data (CMRCVM)" on page 231 provides information about receiving mapped partner data.

Send_Data (CMSEND)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

A program uses the Send_Data (CMSEND) call to send data to the remote program. When issued during a mapped conversation, this call sends one data record to the remote program. The data record consists entirely of data and is not examined by the system for possible logical records.

When issued during a basic conversation, this call sends data to the remote program. The data consists of logical records. The amount of data is specified independently of the data format.

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

CALL CMSST – Set_Send_Type

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

CALL CMSPTR – Set_Prepare_To_Receive_Type

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

CALL CMSDT – Set_Deallocate_Type

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

buffer *(input)*
When a program issues a Send_Data call during a mapped conversation, *buffer* specifies the data record to be sent. The length of the data record is given by the *send_length* parameter.

When a program issues a Send_Data call during a basic conversation, *buffer* specifies the data to be sent. The data consists of logical records, each containing a 2-byte length field (denoted as LL) followed by a data field.

The length of the data field can range from 0 to 32765 bytes. The 2-byte length field contains the following bits:

- A high-order bit that is not examined by the system. It is used, for example, by the system's mapped conversation component in support of the mapped conversation calls.
- A 15-bit binary length of the record.

The length of the record equals the length of the data field plus the 2-byte length field. Therefore, logical record length values of X'0000', X'0001', X'8000', and X'8001' are not valid.

Note: The logical record length values shown above (such as X'0000') are in the hexadecimal (base-16) numbering system.

send_length *(input)*

For both basic and mapped conversations, the *send_length* ranges in value from 0 to the maximum buffer size supported by the system. The maximum buffer size is at least 32767 bytes. See Usage Note 10 on page 257 for additional information about determining the maximum buffer size. The *send_length* parameter specifies the size of the *buffer* parameter and the number of bytes to be sent on the conversation.

When a program issues a Send_Data call during a mapped conversation and *send_length* is zero, a null data record is sent.

When a program issues a Send_Data call during a basic conversation, *send_length* specifies the size of the *buffer* parameter and is **not** related to the length of a logical record. If *send_length* is zero, no data is sent, and the *buffer* parameter is not important. However, the other parameters and setup characteristics are significant and retain their meaning as described.

control_information_received *(output)*

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

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

- CM_NO_CONTROL_INFO_RECEIVED
  Indicates that no control information was received.
- CM_REQ_TO_SEND_RECEIVED (half-duplex conversations only)
  The local program received a request-to-send notification from the remote program. The remote program issued Request_To_Send, requesting the local program's end of the conversation to enter **Receive** state, which would place the remote program's end of the conversation in **Send** state. See "Request_To_Send (CMRTS)" on page 246 for further discussion of the local program's possible responses.
- CM_ALLOCATE_CONFIRMED (OSI TP CRM only)
  The local program received confirmation of the remote program's acceptance of the conversation.
- CM_ALLOCATE_CONFIRMED_WITH_DATA (OSI TP CRM only)
  The local program received confirmation of the remote program's acceptance of the conversation. The local program may now issue an Extract_Initialization_Data (CMEID) call to receive the initialization data.
- CM_ALLOCATE_REJECTED_WITH_DATA (OSI TP CRM only)
  The remote program rejected the conversation. The local program may now issue an Extract_Initialization_Data (CMEID) call to receive the initialization data.

This value will be returned with a return code of CM_OK. The program will receive a CM_DEALLOCATED_ABEND return code on a later call on the conversation.

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

Notes:

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

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

The following return codes apply to half-duplex conversations.

- CM_OK
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_CONVERSATION_TYPE_MISMATCH
- CM_PIP_NOT_SPECIFIED_CORRECTLY
- CM_SECURITY_NOT_VALID
- CM_SYNC_LVL_NOT_SUPPORTED_PGM
- CM_SYNC_LVL_NOT_SUPPORTED_SYS
- CM_SEND_RCV_MODE_NOT_SUPPORTED
- CM_TPN_NOT_RECOGNIZED
- CM_TP_NOT_AVAILABLE_NO_RETRY
- CM_TP_NOT_AVAILABLE_RETRY
- CM_PROGRAM_ERROR_PURGING
- CM_DEALLOCATED_ABEND
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_DEALLOCATED_ABEND_SVC (basic conversations only)
- CM_DEALLOCATED_ABEND_TIMER (basic conversations only)
- CM_SVC_ERROR_PURGING (basic conversations only)
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  – The conversation is not in **Send**, **Send-Pending**, **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** state.

- The conversation is in **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** state, and the program received a take-commit notification not ending in *_DATA_OK.
- The conversation is basic and in **Send** state; the *send_type* is set to CM_SEND_AND_CONFIRM, CM_SEND_AND_DEALLOCATE, or CM_SEND_AND_PREP_TO_RECEIVE; the *deallocate_type* is not set to CM_DEALLOCATE_ABEND (if *send_type* is set to CM_SEND_AND_DEALLOCATE); and the data does not end on a logical record boundary.
- For a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the conversation's context is in the **Backout-Required** condition. The Send_Data call is not allowed for this conversation while its context is in this condition.
- The conversation is in **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** state, and *send_type* is set to CM_SEND_AND_CONFIRM or CM_SEND_AND_PREP_TO_RECEIVE.
- The conversation is in **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** state, the *send_type* is set to CM_SEND_AND_DEALLOCATE, and the *deallocate_type* is not set to CM_DEALLOCATE_ABEND.
- The *send_type* is set to CM_SEND_AND_DEALLOCATE and the following conditions are also true:
  - The *deallocate_type* is set to CM_DEALLOCATE_FLUSH or CM_DEALLOCATE_CONFIRM.
  - The *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.
  - The conversation is included in a transaction.

- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *send_length* exceeds the range permitted by the implementation. The maximum value of the length in each implementation is at least 32767. See Usage Note 10 on page 257 for additional information about determining the maximum buffer size.
  - The *conversation_type* is CM_BASIC_CONVERSATION and *buffer* contains an invalid logical record length (LL) value of X'0000', X'0001', X'8000', or X'8001'.
  - The *send_type* is set to CM_SEND_AND_PREP_TO_RECEIVE and the following conditions are also true:
    - The *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL.
    - The *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.
    - The conversation is included in a transaction.
    - The conversation is using an OSI TP CRM, and the program is not the superior for the conversation.
  - The *send_type* is set to CM_SEND_AND_DEALLOCATE and the following conditions are also true:
    - The *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL.
    - The *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.
    - The conversation is included in a transaction.
    - The conversation is using an OSI TP CRM, and the program is not the superior for the conversation.

- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR
- The following values are returned only when *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and the conversation is included in a transaction:
  - CM_TAKE_BACKOUT
  - CM_DEALLOCATED_ABEND_BO
  - CM_DEALLOCATED_ABEND_SVC_BO (basic conversations only)
  - CM_DEALLOCATED_ABEND_TIMER_BO (basic conversations only)
  - CM_RESOURCE_FAIL_NO_RETRY_BO
  - CM_RESOURCE_FAILURE_RETRY_BO
  - CM_INCLUDE_PARTNER_REJECT_BO

The following return codes apply to full-duplex conversations.

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

- CM_OK
- CM_ALLOCATION_ERROR
- CM_DEALLOCATED_ABEND
- CM_DEALLOCATED_ABEND_SVC
- CM_DEALLOCATED_ABEND_TIMER
- CM_DEALLOCATED_CONFIRM_REJECT
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_DEALLOCATED_NORMAL (OSI TP CRM only)
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The conversation is not in **Send-Receive**, **Send-Only**, **Sync-Point**, or **Sync-Point-Deallocate** state.
  - The conversation is basic and in **Send-Receive** or **Send-Only** state, the *send_type* is set to CM_SEND_AND_DEALLOCATE, the *deallocate_type* is not set to CM_DEALLOCATE_ABEND, and the data does not end on a logical record boundary.
  - The conversation is in **Sync-Point** or **Sync-Point Deallocate** state and the program received a take-commit notification not ending in *_DATA_OK.
  - The conversation is in **Sync-Point**, or **Sync-Point-Deallocate** state, the *send_type* is set to CM_SEND_AND_DEALLOCATE, and the *deallocate_type* is not set to CM_DEALLOCATE_ABEND.
  - The *send_type* is set to CM_SEND_AND_DEALLOCATE and the following conditions are also true:
    - The *deallocate_type* is set to CM_DEALLOCATE_FLUSH or CM_DEALLOCATE_CONFIRM.
    - The *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM.
    - The conversation is included in a transaction.
  - For a conversation with *sync_level* set to CM_SYNC_POINT_NO_CONFIRM, this return code indicates one of the following:
    - The conversation's context is in the **Backout-Required** condition. The Send_Data call is not allowed for this conversation while its context is in this condition.

- The local program has received a *status_received* value of CM_JOIN_TRANSACTION and must issue a tx_begin call to the X/Open TX interface to join the transaction.

- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  – The *conversation_ID* specifies an unassigned conversation identifier.
  – The *send_length* exceeds the range permitted by the implementation. The maximum value of the length in each implementation is at least 32767.
  – The *conversation_type* is CM_BASIC_CONVERSATION and *buffer* contains an invalid logical record length (LL) value of X'0000', X'0001', X'8000', or X'8001'.
  – The *send_type* is set to CM_SEND_AND_DEALLOCATE and the following conditions are also true:
    - The *deallocate_type* is set to CM_DEALLOCATE_FLUSH or CM_DEALLOCATE_CONFIRM.
    - The *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM.
    - The conversation in included in a transaction.
  – The *send_type* is set to CM_SEND_AND_DEALLOCATE and the following conditions are also true:
    - The *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL.
    - The *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM.
    - The conversation is included in a transaction.
    - The program is not the superior for the conversation.
- CM_OPERATION_NOT_ACCEPTED
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_PRODUCT_SPECIFIC_ERROR

The following values are returned only if *sync_level* is CM_SYNC_POINT_NO_CONFIRM, the state is **Send-Receive** and the conversation is currently included in a transaction.

- CM_TAKE_BACKOUT
- CM_DEALLOCATED_ABEND_BO
- CM_DEALLOCATED_ABEND_SVC_BO (basic conversations only)
- CM_DEALLOCATED_ABEND_TIMER_BO (basic conversations only)
- CM_RESOURCE_FAIL_NO_RETRY_BO
- CM_RESOURCE_FAILURE_RETRY_BO
- CM_CONV_DEALLOC_AFTER_SYNCPT
- CM_INCLUDE_PARTNER_REJECT_BO

State Changes

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

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

  – *Prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_FLUSH
  – *Prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_CONFIRM
  – *Prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* is set to CM_NONE or CM_CONFIRM

  - *Prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, but the conversation is not currently included in a transaction.

- The conversation enters **Defer-Receive** state when Send_Data is issued with *send_type* set to CM_SEND_AND_PREP_TO_RECEIVE, *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and the conversation is included in a transaction.
- The conversation enters **Reset** state when Send_Data is issued with *send_type* set to CM_SEND_AND_DEALLOCATE and any of the following conditions is true:
  - *Deallocate_type* is set to CM_DEALLOCATE_ABEND
  - *Deallocate_type* is set to CM_DEALLOCATE_FLUSH
  - *Deallocate_type* is set to CM_DEALLOCATE_CONFIRM
  - *Deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is set to CM_NONE or CM_CONFIRM
  - *Deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, but the conversation is not currently included in a transaction.
- The conversation enters **Defer-Deallocate** state when Send_Data is issued with *send_type* set to CM_SEND_AND_DEALLOCATE, *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and the conversation is included in a transaction.
- The conversation enters **Send** state when Send_Data is issued in **Send-Pending** state with *send_type* set to CM_BUFFER_DATA, CM_SEND_AND_FLUSH, or CM_SEND_AND_CONFIRM.
- No state change occurs when Send_Data is issued in **Send** state with *send_type* set to CM_BUFFER_DATA, CM_SEND_AND_FLUSH, or CM_SEND_AND_CONFIRM.

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

- The conversation enters **Receive-Only** state when the Send_Data call is issued in **Send-Receive** state with *send_type* set to CM_SEND_AND_DEALLOCATE, and one of the following conditions is true:
  - *deallocate_type* is set to CM_DEALLOCATE_FLUSH
  - *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is set to CM_NONE
  - *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, and the conversation is not currently included in a transaction.
- The conversation enters **Reset** state when the Send_Data call is issued with *send_type* set to CM_SEND_AND_DEALLOCATE and one of the following conditions is true:
  - The call is issued in **Send-Only** state
  - The call is issued in **Send-Receive**, **Send-Only**, **Sync-Point**, or **Sync-Point-Deallocate** state and *deallocate_type* is set to CM_DEALLOCATE_ABEND.
- The conversation enters **Defer-Deallocate** state when the Send_Data call is issued with *send_type* set to CM_SEND_AND_DEALLOCATE, *deallocate_type* set

to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* set to CM_SYNC_POINT_NO_CONFIRM, and the conversation included in a transaction.

- No state change occurs when Send_Data is issued in **Send-Receive** or **Send-Only** state with *send_type* set to CM_BUFFER_DATA or CM_SEND_AND_FLUSH.

Usage Notes

1. The local system buffers the data to be sent to the remote system until it accumulates a sufficient amount of data for transmission (from one or more Send_Data calls), or until the local program issues a call that causes the system to flush its send buffer. The amount of data sufficient for transmission depends on the characteristics of the logical connection allocated for the conversation, and varies from one logical connection to another.

2. For a half-duplex conversation, when *control_information_received* indicates CM_REQ_TO_SEND_RECEIVED, or CM_RTS_RCVD_AND_EXP_DATA_AVAIL, or the remote program is requesting the local program's end of the conversation to enter **Receive** state, which places the remote program's end of the conversation in **Send** state. See “Request_To_Send (CMRTS)” on page 246 for a discussion of how a program can place its end of a conversation in **Receive** state.

3. When issued during a mapped conversation, the Send_Data call sends one complete data record. The data record consists entirely of data, and CPI Communications does not examine the data for logical record length fields. It is this complete data record that is indicated to the remote program by the *data_received* parameter of the Receive call.

   For example, consider a mapped conversation where the local program issues two Send_Data calls with *send_length* set, respectively, to 30 and then 50. (These numbers are simplistic for explanatory purposes.) The local program then issues Flush and the 80 bytes of data are sent to the remote system. The remote program now issues Receive with *requested_length* set to a sufficiently large value, say 1000. The remote program will receive back only 30 bytes of data (indicated by the *received_length* parameter) because this is a complete data record. The completeness of the data record is indicated by the *data_received* variable, which will be set to CM_COMPLETE_DATA_RECEIVED.

   The remote program receives the remaining 50 bytes of data (from the second Send_Data) when it performs a second Receive with *requested_length* set to a value greater than or equal to 50.

4. The data sent by the program during a basic conversation consists of logical records. The logical records are independent of the length of data as specified by the *send_length* parameter. The data can contain one or more complete records, the beginning of a record, the middle of a record, or the end of a record. The following combinations of data are also possible:

   - One or more complete records, followed by the beginning of a record
   - The end of a record, followed by one or more complete records
   - The end of a record, followed by one or more complete records, followed by the beginning of a record
   - The end of a record, followed by the beginning of a record

5. The program using a basic conversation must finish sending a logical record before issuing any of the following calls:

- Confirm
- Deallocate with *deallocate_type* set to CM_DEALLOCATE_FLUSH, CM_DEALLOCATE_CONFIRM, or CM_DEALLOCATE_SYNC_LEVEL
- Include_Partner_In_Transaction
- Prepare
- Prepare_To_Receive
- Receive
- Resource recovery commit

A program finishes sending a logical record when it sends a complete record or when it truncates an incomplete record. The data must end with the end of a logical record (on a logical record boundary) when Send_Data is issued with *send_type* set to CM_SEND_AND_CONFIRM, CM_SEND_AND_DEALLOCATE, or CM_SEND_AND_PREP_TO_RECEIVE.

6. A complete logical record contains the 2-byte LL field and all bytes of the data field, as determined by the logical-record length. If the data field length is zero, the complete logical record contains only the 2-byte length field. An incomplete logical record consists of any amount of data less than a complete record. It can consist of only the first byte of the LL field, the 2-byte LL field plus all of the data field except the last byte, or any amount in between. A logical record is incomplete until the last byte of the data field is sent, or until the second byte of the LL field is sent if the data field is of zero length.

7. During a basic conversation, a program can truncate an incomplete logical record by issuing the Send_Error call. Send_Error causes the system to flush its send buffer, which includes sending the truncated record. The system then treats the first two bytes of data specified in the next Send_Data as the LL field. Issuing Send_Data with *send_type* set to CM_SEND_AND_DEALLOCATE and *deallocate_type* set to CM_DEALLOCATE_ABEND, or Deallocate with *deallocate_type* set to CM_DEALLOCATE_ABEND, during a basic conversation also truncates an incomplete logical record. If the *log_data* characteristic is not null and these conditions occur, log data is sent.

8. Send_Data is often used in combination with other calls, such as Flush, Confirm, and Prepare_To_Receive. Contrast this usage with the equivalent function available from the use of the Set_Send_Type call prior to issuing a call to Send_Data.

9. When a Send_Data call is issued with *send_type* set to CM_SEND_AND_DEALLOCATE, *deallocate_type* set to CM_DEALLOCATE_ABEND, and *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the conversation's context may be placed in the **Backout-Required** condition.

10. A program must not specify a value in the *send_length* parameter that is greater than the maximum the implementation can support. The maximum may vary from system to system. The program can use the Extract_Maximum_Buffer_Size call to find out the maximum buffer size supported by the local system. The program can achieve portability across different systems by using one of the following methods:

    a. Never using a *send_length* value greater than 32767

    b. Using the Extract_Maximum_Buffer_Size call to determine the maximum buffer size supported by the system and never setting *send_length* greater than that maximum buffer size

11. When *control_information_received* indicates that expedited data is available to be received, subsequent calls with this parameter will continue to indicate that

expedited data is available until the expedited data has been received by the program.

Related Information

"Conversation Types" on page 19 provides more information on mapped and basic conversations.

"Data Buffering and Transmission" on page 44 provides a complete discussion of controls over data transmission.

All of the example program flows in Chapter 3, "Program-to-Program Communication Example Flows" make use of the Send_Data call.

"Extract_Maximum_Buffer_Size (CMEMBS)" on page 173 further discusses determining the maximum buffer size supported by the system.

"Receive (CMRCV)" on page 213 provides more information on the *data_received* parameter.

"Receive_Mapped_Data (CMRCVM)" on page 231 provides more information on receiving mapped partner data.

"Send_Mapped_Data (CMSNDM)" on page 271 provides more information on sending mapped partner data.

"Set_Send_Type (CMSST)" on page 351 provides more information on the *send_type* conversation characteristic and the use of it in combination with calls to Send_Data.

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

Send_Error (CMSERR)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

Send_Error (CMSERR) is used by a program to inform the remote program that the local program detected an error during a conversation. If the conversation is in **Send**, **Send-Receive**, or **Send-Only** state, Send_Error forces the system to flush its send buffer.

For a half-duplex conversation, when this call completes successfully, the local program's end of the conversation is in **Send** state and the remote program's end of the conversation is in **Receive** state. Further action is defined by program logic. Typically, this involves sending information about the error to the partner.

For a full-duplex conversation, no state change occurs. The issuance of Send_Error will be reported to the partner on a Receive call.

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

CALL CMSED – Set_Error_Direction (for half-duplex conversations only)
CALL CMSLD – Set_Log_Data

Format

CALL CMSERR(*conversation_ID*,
control_information_received,
return_code)

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

control_information_received *(output)*
Specifies the variable containing an indication of whether or not control information has been received.

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

- CM_NO_CONTROL_INFO_RECEIVED
  Indicates that no control information was received.
- CM_REQ_TO_SEND_RECEIVED (half-duplex conversations only)
  The local program received a request-to-send notification from the remote program. The remote program issued Request_To_Send, requesting the local program's end of the conversation to enter **Receive** state, which would place the remote program's end of the conversation in **Send** state. See "Request_To_Send (CMRTS)" on page 246 for further discussion of the local program's possible responses.

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

  This value will be returned with a return code of CM_OK. The program will receive a CM_DEALLOCATED_ABEND return code on a later call on the conversation.
- CM_EXPEDITED_DATA_AVAILABLE (LU 6.2 CRM only)
  Expedited data is available to be received.
- CM_RTS_RCVD_AND_EXP_DATA_AVAIL (half-duplex conversations and LU 6.2 CRM only)
  The local program received a request-to-send notification from the remote program and expedited data is available to be received.

Notes:

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

return_code *(output)*
Specifies the result of the call execution. The value for *return_code* depends on the state of the conversation at the time this call is issued.

The following return codes apply to half-duplex conversations.

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

- CM_OK
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_CONVERSATION_TYPE_MISMATCH
- CM_PIP_NOT_SPECIFIED_CORRECTLY
- CM_SECURITY_NOT_VALID
- CM_SYNC_LVL_NOT_SUPPORTED_PGM
- CM_SYNC_LVL_NOT_SUPPORTED_SYS
- CM_SEND_RCV_MODE_NOT_SUPPORTED

- CM_TPN_NOT_RECOGNIZED
- CM_TP_NOT_AVAILABLE_NO_RETRY
- CM_TP_NOT_AVAILABLE_RETRY
- CM_DEALLOCATED_ABEND
- CM_PROGRAM_ERROR_PURGING
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_DEALLOCATED_ABEND_SVC (basic conversations only)
- CM_DEALLOCATED_ABEND_TIMER (basic conversations only)
- CM_SVC_ERROR_PURGING (basic conversations only)
- CM_PROGRAM_PARAMETER_CHECK
  The *conversation_ID* specifies an unassigned conversation identifier.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR
- The following values are returned only when *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and the conversation is included in a transaction:
  - CM_TAKE_BACKOUT
  - CM_DEALLOCATED_ABEND_BO
  - CM_DEALLOCATED_ABEND_SVC_BO (basic conversations only)
  - CM_DEALLOCATED_ABEND_TIMER_BO (basic conversations only)
  - CM_PROGRAM_STATE_CHECK
    This return code indicates that for a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the conversation's context is in the **Backout-Required** condition. The Send_Error call is not allowed for this conversation while its context is in this condition.
  - CM_RESOURCE_FAIL_NO_RETRY_BO
  - CM_RESOURCE_FAILURE_RETRY_BO
  - CM_INCLUDE_PARTNER_REJECT_BO

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

- CM_OK
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_SYNC_LVL_NOT_SUPPORTED_SYS
- CM_SEND_RCV_MODE_NOT_SUPPORTED
- CM_TPN_NOT_RECOGNIZED
- CM_TP_NOT_AVAILABLE_NO_RETRY
- CM_TP_NOT_AVAILABLE_RETRY
- CM_DEALLOCATED_ABEND
- CM_PROGRAM_ERROR_PURGING
- CM_DEALLOCATED_NORMAL
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_PROGRAM_PARAMETER_CHECK
  The *conversation_ID* specifies an unassigned conversation identifier.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR
- The following values are returned only when *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and the conversation is included in a transaction:
  - CM_TAKE_BACKOUT
  - CM_DEALLOCATED_ABEND_BO

  - CM_DEALLOCATED_NORMAL_BO
  - CM_PROGRAM_STATE_CHECK
    This return code indicates one of the following:
    - For a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the conversation's context is in the **Backout-Required** condition. The Send_Error call is not allowed for this conversation while its context is in this condition.
    - The local program has received a *status_received* value of CM_JOIN_TRANSACTION and must issue a tx_begin call to the X/Open TX interface to join the transaction.
  - CM_RESOURCE_FAIL_NO_RETRY_BO
  - CM_RESOURCE_FAILURE_RETRY_BO
  - CM_INCLUDE_PARTNER_REJECT_BO

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

- CM_OK
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_PROGRAM_PARAMETER_CHECK
  The *conversation_ID* specifies an unassigned conversation identifier.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR
- The following values are returned only when *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM:
  - CM_TAKE_BACKOUT
  - CM_PROGRAM_STATE_CHECK
    This return code indicates that for a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the conversation's context is in the **Backout-Required** condition. The Send_Error call is not allowed for this conversation while its context is in this condition.
  - CM_RESOURCE_FAIL_NO_RETRY_BO
  - CM_RESOURCE_FAILURE_RETRY_BO
  - CM_INCLUDE_PARTNER_REJECT_BO

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

- CM_OK
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_PROGRAM_PARAMETER_CHECK
  The *conversation_ID* specifies an unassigned conversation identifier.
- CM_PROGRAM_STATE_CHECK
  This return code indicates that for a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the conversation's context is in the **Backout-Required** condition. The Send_Error call is not allowed for this conversation while its context is in this condition.
- CM_TAKE_BACKOUT

This value is returned only when the Send_Error call is issued in **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** state, and only when the conversation is using an OSI TP CRM.
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_RESOURCE_FAIL_NO_RETRY_BO
  This value is returned only when *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.
- CM_RESOURCE_FAILURE_RETRY_BO
  This value is returned only when *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.
- CM_TAKE_BACKOUT
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

Otherwise, the conversation is in **Reset**, **Initialize**, **Defer-Receive**, **Defer-Deallocate**, **Initialize-Incoming**, or **Prepared** state and *return_code* has one of the following values:

- CM_PROGRAM_PARAMETER_CHECK
  The *conversation_ID* specifies an unassigned identifier.
- CM_OPERATION_NOT_ACCEPTED
- CM_PROGRAM_STATE_CHECK

The following return codes apply to full-duplex conversations.

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

- CM_OK
- CM_ALLOCATION_ERROR
- CM_DEALLOCATED_ABEND
- CM_DEALLOCATED_ABEND_SVC
- CM_DEALLOCATED_ABEND_TIMER
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_DEALLOCATED_NORMAL (OSI TP CRM only)
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  – The conversation is not in **Send-Receive**, **Send-Only**, or **Confirm-Deallocate** state.
  – The local program has received a *status_received* value of CM_JOIN_TRANSACTION and must issue a tx_begin call to the X/Open TX interface to join the transaction.
  – For a conversation with *sync_level* set to CM_SYNC_POINT_NO_CONFIRM, the conversation's context is in the **Backout-Required** condition. The Send_Error call is not allowed for this conversation while its context is in this condition.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates the *conversation_ID* specifies an unassigned conversation identifier.
- CM_OPERATION_NOT_ACCEPTED
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_PRODUCT_SPECIFIC_ERROR

- The following values are returned only if *sync_level* is CM_SYNC_POINT_NO_CONFIRM, the state is **Send-Receive**, and the conversation is included in a transaction.
  - CM_TAKE_BACKOUT
  - CM_DEALLOCATED_ABEND_BO
  - CM_DEALLOCATED_ABEND_SVC_BO (basic conversations only)
  - CM_DEALLOCATED_ABEND_TIMER_BO (basic conversations only)
  - CM_RESOURCE_FAIL_NO_RETRY_BO
  - CM_RESOURCE_FAILURE_RETRY_BO
  - CM_CONV_DEALLOC_AFTER_SYNCPT
  - CM_INCLUDE_PARTNER_REJECT_BO

State Changes

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

- The conversation enters **Send** state when the call is issued in **Receive**, **Confirm**, **Confirm-Send**, **Confirm-Deallocate**, or **Send-Pending** state. For a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and which is included in a transaction, the conversation also enters **Send** state when the call is issued in **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** state.
- No state change occurs when the call is issued in **Send** state.

For full-duplex conversations, when *return_code* indicates CM_OK, the conversation enters **Send-Receive** state when this call is issued in **Confirm-Deallocate** state.

Usage Notes

1. The system can send the error notification to the remote system immediately (during the processing of this call), or the system can delay sending the notification until a later time. If the system delays sending the notification, it buffers the notification until it has accumulated a sufficient amount of information for transmission, or until the local program issues a call that causes the system to flush its send buffer.
2. The amount of information sufficient for transmission depends on the characteristics of the logical connection allocated for the conversation, and varies from one logical connection to another. Transmission of the information can begin immediately if the *log_data* characteristic has been specified with sufficient log data, or transmission can be delayed until sufficient data from subsequent Send_Data calls is also buffered.
3. To make sure that the remote program receives the error notification as soon as possible, the local program can issue Flush immediately after Send_Error.
4. For a half-duplex conversation using an LU 6.2 CRM, the issuance of Send_Error is reported to the remote program as one of the following return codes:
   - CM_PROGRAM_ERROR_TRUNC (basic conversation)
     The local program issued Send_Error with its end of the conversation in **Send** state after sending an incomplete logical record (see "Send_Data (CMSEND)" on page 249). The record has been truncated.
   - CM_PROGRAM_ERROR_NO_TRUNC (basic and mapped conversations)
     The local program issued Send_Error with its end of the conversation in **Send** state after sending a complete logical record (basic) or data record

(mapped); or before sending any record; or the local program issued Send_Error with its end of the conversation in **Send-Pending** state with *error_direction* set to CM_SEND_ERROR. No truncation has occurred.

- CM_PROGRAM_ERROR_PURGING (basic and mapped conversations)
  The local program issued Send_Error with its end of the conversation in **Receive** state, and all information sent by the remote program and not yet received by the local program has been purged. Or the local program issued Send_Error with its end of the conversation in **Send-Pending** state and *error_direction* set to CM_RECEIVE_ERROR or in **Confirm**, **Confirm-Send**, or **Confirm-Deallocate** state, and no purging has occurred.

5. If the conversation is using an OSI TP CRM, the remote program receives CM_PROGRAM_ERROR_PURGING, regardless of the conversation state.

6. When a half-duplex conversation is using an LU 6.2 CRM and Send_Error is issued in **Receive** state, incoming information is also purged. Because of this purging, the *return_code* of CM_DEALLOCATED_NORMAL is reported instead of:

   - CM_CONVERSATION_TYPE_MISMATCH
   - CM_PIP_NOT_SPECIFIED_CORRECTLY
   - CM_SECURITY_NOT_VALID
   - CM_SYNC_LVL_NOT_SUPPORTED_PGM
   - CM_TPN_NOT_RECOGNIZED
   - CM_TP_NOT_AVAILABLE_NO_RETRY
   - CM_TP_NOT_AVAILABLE_RETRY
   - CM_DEALLOCATED_ABEND
   - CM_DEALLOCATED_ABEND_SVC (basic conversations only)
   - CM_DEALLOCATED_ABEND_TIMER (basic conversations only)

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

   - CM_DEALLOCATED_ABEND_BO
   - CM_DEALLOCATED_ABEND_SVC_BO (basic conversations only)
   - CM_DEALLOCATED_ABEND_TIMER_BO (basic conversations only)

   Similarly, a return code of CM_OK is reported instead of:

   - CM_PROGRAM_ERROR_NO_TRUNC
   - CM_PROGRAM_ERROR_PURGING
   - CM_PROGRAM_ERROR_TRUNC (basic conversations only)
   - CM_SVC_ERROR_NO_TRUNC (basic conversations only)
   - CM_SVC_ERROR_PURGING (basic conversations only)
   - CM_SVC_ERROR_TRUNC (basic conversations only)
   - CM_TAKE_BACKOUT

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

   The following types of incoming information are also purged:

   - Data sent with the Send_Data call.
   - Confirmation request sent with the Send_Data, Confirm, Prepare_To_Receive, or Deallocate call.
     If the confirmation request was sent with *deallocate_type* set to CM_DEALLOCATE_CONFIRM or CM_DEALLOCATE_SYNC_LEVEL, the deallocation request will also be purged.
   - Prepare call or resource recovery commit call.

If the Prepare or commit call was sent in conjunction with a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL, the deallocation request will also be purged.

The request-to-send notification is not purged. This notification is reported to the program when it issues a call that includes the *control_information_received* parameter.

7. The program can use this call for various application-level functions. For example, the program can issue this call to truncate an incomplete logical record it is sending; to inform the remote program of an error detected in data received; or to reject a confirmation request.

8. If the *log_data_length* characteristic is greater than zero, the system formats the supplied log data into the appropriate format. The data supplied by the program is any data the program wants to have logged. The data is logged on the local system's error log and is also sent to the remote system for logging there.

   The *log_data* is not sent on the Send_Error call when an OSI TP CRM is being used for the conversation. Instead, it is ignored.

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

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

   Networking Services for Windows does not send *log_data* to the partner program's system and does not log data associated with outgoing and incoming Send_Error and incoming Deallocate calls. *log_data* is placed in the trace log when the Deallocate call is issued.

9. The *error_direction* characteristic is significant only when a half-duplex conversation is using an LU 6.2 CRM and Send_Error is issued in **Send-Pending** state (that is, the Send_Error is issued immediately following a Receive on which both data and a *status_received* parameter set to CM_SEND_RECEIVED is received). In this case, Send_Error could be reporting one of the following types of errors:

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

   Because the system cannot tell which of the two errors occurred, the program has to supply the *error_direction* information.

   The default for *error_direction* is CM_RECEIVE_ERROR. A program can override the default using the Set_Error_Direction call before issuing Send_Error.

   Once changed, the new *error_direction* value remains in effect until the program changes it again. Therefore, a program should issue Set_Error_Direction before issuing Send_Error for a conversation in **Send-Pending** state.

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

10. When *control_information_received* indicates that expedited data is available, subsequent calls with this parameter will continue to indicate that expedited data is available until the expedited data has been received by the program.

11. For full-duplex conversations, the issuance of Send_Error is reported on the remote program's Receive call as one of the following return codes:

    - CM_PROGRAM_ERROR_NO_TRUNC (basic and mapped conversations using an LU 6.2 CRM)
    - CM_PROGRAM_ERROR_TRUNC (basic conversations using an LU 6.2 CRM)
    - CM_PROGRAM_ERROR_PURGING (conversations using an OSI TP CRM)

    No data is purged, unless the conversation is using an OSI TP CRM, in which case, the program should expect purging. The partner program may expect the CM_PROGRAM_ERROR_PURGING return code if the conversation is allocated using an OSI TP CRM. The programs may know whether to expect purging by issuing an Extract_Partner_ID call.

12. Send_Error does not complete successfully if an error that causes the conversation to terminate has occurred or the remote program has issued a Cancel_Conversation call, a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND, or a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_FLUSH and the conversation has been allocated using an OSI TP CRM.

    For a conversation which is not included in a transaction, a CM_DEALLOCATED_ABEND_*, CM_ALLOCATION_ERROR, CM_RESOURCE_FAILURE_*_RETRY, or CM_DEALLOCATED_NORMAL return code is returned. When one of the above return codes is returned and the conversation is in **Send-Receive** state, the program can terminate the conversation by issuing Receives until it gets one of the above return codes taking it to **Reset** state, or by issuing Cancel_Conversation or Deallocate with *deallocate_type* set to CM_DEALLOCATE_ABEND.

    For a conversation which is included in a transaction, CM_DEALLOCATED_ABEND_*_BO, CM_ALLOCATION_ERROR, CM_RESOURCE_FAILURE_RETRY_BO, or CM_RESOURCE_FAIL_NO_RETRY_BO is returned. If CM_ALLOCATION_ERROR is returned, the program behaves as though it were not in transaction, otherwise it is in **Backout-Required** condition and in **Reset** state.

Related Information

"Example 6: Reporting Errors" on page 80 and "Example 7: Error Direction and Send-Pending State" on page 82 provide example program flows using Send_Error and the **Send-Pending** state; "Set_Error_Direction (CMSED)" on page 307 provides further information on the *error_direction* characteristic.

"Usage Notes" of "Request_To_Send (CMRTS)" on page 247 provides more information on how a conversation enters **Receive** state.

"Send_Data (CMSEND)" on page 249 discusses basic conversations and logical records.

"Set_Log_Data (CMSLD)" on page 316 provides a description of the *log_data* characteristic.

Send_Expedited_Data (CMSNDX)

LU 6.2

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
					X*			

A program uses the Send_Expedited_Data (CMSNDX) call to send expedited data to its partner.

This call has meaning only when an LU 6.2 CRM is used for the conversation.

X* In OS/2, this call is supported in Communications Server.

Format

```
CALL CMSNDX(conversation_ID,
            buffer,
            send_length,
            control_information_received,
            return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

buffer *(input)*
Specifies the variable containing the data to be sent.

send_length *(input)*
Specifies the length of the data to be sent. The minimum amount of data that can be sent is 1 byte; the maximum is 86 bytes.

control_information_received *(output)*
Specifies the variable containing an indication of whether or not control information has been received.

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

- CM_NO_CONTROL_INFO_RECEIVED
  Indicates that no control information was received.
- CM_REQ_TO_SEND_RECEIVED (half-duplex conversations only)
  The local program received a request-to-send notification from the remote program. The remote program issued Request_To_Send, requesting the local program's end of the conversation to enter **Receive** state, which would place the remote program's end of the conversation in **Send** state. See “Request_To_Send (CMRTS)” on page 246 for further discussion of the local program's possible responses.
- CM_EXPEDITED_DATA_AVAILABLE
  Expedited data is available to be received.
- CM_RTS_RCVD_AND_EXP_DATA_AVAIL (half-duplex conversations only)

The local program received a request-to-send notification from the remote program and expedited data is available to be received.

Notes:

1. If *return_code* is set to CM_PROGRAM_PARAMETER_CHECK or CM_PROGRAM_STATE_CHECK, the value contained in *control_information_received* has no meaning.

2. When more than one piece of control information is available to be returned to the program, it will be returned in the following order:

   - CM_RTS_RCVD_AND_EXP_DATA_AVAIL
   - CM_REQ_TO_SEND_RECEIVED
   - CM_EXPEDITED_DATA_AVAILABLE
   - CM_NO_CONTROL_INFO_RECEIVED

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_PROGRAM_PARAMETER_CHECK
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *send_length* specifies a value less than 1 or greater than 86.
  - The conversation is not using an LU 6.2 CRM.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is in **Initialize** or **Initialize-Incoming** state and is not allowed to send expedited data.
- CM_CONVERSATION_ENDING
  This value indicates that the conversation is ending due to a normal deallocation, an allocation error, a Cancel_Conversation call, a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND, or a conversation failure. Hence, no expedited data is sent.
- CM_EXP_DATA_NOT_SUPPORTED
  This value indicates that the remote system does not support expedited data.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. A program uses the Send_Expedited_Data call to send data that flows in an expedited fashion, possibly bypassing data sent using the Send_Data call.

2. The Send_Expedited_Data call should be used sparingly and should not be used for sending normal data.

3. When the remote system receives the expedited data, it retains the expedited data until it is received by the partner program using Receive_Expedited_Data.

4. **Note to Implementers:** A *control_information_received* notification can be reported on this call (associated with the Expedited-Send queue), on the

Receive_Expedited_Data call (associated with the Expedited-Receive queue), on the Send_Data call (associated with the Send queue or the Send-Receive queue), on the Receive call (associated with the Receive queue or the Send-Receive queue), and on the Test_Request_To_Send_Received call (not associated with any queue). When the program uses multiple threads or queue-level non-blocking, more than one of these calls may be executed simultaneously. An implementation should report the CM_EXPEDITED_DATA_AVAILABLE indication to the program through all available calls, until the expedited data is received. All other values of *control_information_received* should be reported only once.

Related Information

"Receive_Expedited_Data (CMRCVX)" on page 228 describes the Receive_Expedited_Data call.

Send_Mapped_Data (CMSNDM)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32

A program issues the Send_Mapped_Data (CMSNDM) call to send data that requires some type of transformation, for example, ASN.1 encoding, before it is sent to the remote partner. The *map_name* is used to identify to the local map routine what data is being sent so that it can apply the correct transforms.

When issued during a basis conversation, this call results in an error.

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

CALL CMSST - Set_Send_Type

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

CALL CMSPTR - Set_Prepare_To_Receive_Type

If *send type*=**CM_SEND_AND_DEALLOCATE,** optional set may include:

CALL CMSDT - Set_Deallocate_Type

Format

```
CALL CMSNDM(conversation_ID,
            map_name,
            map_name_length,
            buffer,
            send_length,
            control_information_received,
            return_code)
```

Parameters

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

map_name *(input)*
Specifies the mapping function used to encode the data record. This is a 0 to 64 byte identifier.

map_name_length *(input)*
Specifies the length of the map_name. The length can be from 0 to 64 bytes. If the length is 0, the *map_name* is ignored.

buffer *(input)*
Specifies the data record to be sent. The length of the data record is given by the *send_length* parameter.

send_length (input)
The *send_length* ranges in value from 0 to the maximum buffer size supported by the system. The maximum buffer size depends on the map name. See Usage Note 6 on page 257 for additional information about determining the maximum buffer size. The *send_length* parameter specifies the size of the *buffer* parameter and the number of bytes to be sent to the local map routine. The map routine will calculate the length of the data to be sent on the conversation and handle it in accordance with the underlying CRM.

When a program issues a Send_Mapped_Data call and *send_length* is 0, a null data record is sent. The map routine will determine if a null record will be sent on the conversation. Refer to site documentation for that particular map routine.

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

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

- CM_NO_CONTROL_INFO_RECEIVED
  Indicates that no control information was received.
- CM_REQ_TO_SEND_RECEIVED (half-duplex conversations only)
  The local program received a request-to-send notification from the remote program. The remote program issued Request_To_Send, requesting the local program's end of the conversation to enter **Receive** state, which would place the remote program's end of the conversation in **Send** state. See "Request_To_Send (CMRTS)" on page 246 for further information about the local program's possible responses.
- CM_ALLOCATE_CONFIRMED (OSI TP CRM only) The local program received confirmation of the remote program's acceptance of the conversation.
- CM_ALLOCATE_CONFIRMED_WITH_DATA (OSI TP CRM only) The local received confirmation of the remote program's acceptance of the conversation. The local program received confirmation of the remote program's acceptance of the conversation. The local program may now issue an Extract_Mapped_Initialization_Data (CMEMID) call to receive the initialization data.

  data received_*(output)*
- CM_ALLOCATE_REJECTED_WITH_DATA (OSI TP CRM only)
  The remote program rejected the conversation. The local program may now issue an Extract_Mapped_Initialization_Data (CMEMID) call to receive the initialization data.
  This value will be returned with a return code of **CM_OK**. The program will receive a CM_DEALLOCATED_ABEND return code on a later call on the conversation.
- CM_EXPEDITED_DATA_AVAILABLE (LU 6.2 CRM only)
  Expedited data is available to be received.
- CM_RTS_RCVD_AND_EXP_DATA_AVAIL (half-duplex conversations and LU 6.2 CRM only)
  The local program received a request-to-send notification form the remote program and expedited data is available to be received.
- CM_EXPEDITED_MAPPED_DATA_AVAILABLE (LU 6.2 CRM only)

Mapped expedited data is available to be received.

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

Notes:

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

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

The following return codes apply to half-duplex conversations.

- CM_OK
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_CONVERSATION_TYPE_MISMATCH
- CM_CONVERSATION_TYPE_MISMATCH
- CM_PIP_NOT_SPECIFIED_CORRECTLY
- CM_SECURITY_NOT_VALID
- CM_SYNC_LVL_NOT_SUPPORTED_PGM
- CM_SYNC_LVL_NOT_SUPPORTED_SYS
- CM_TPN_NOT_RECOGNIZED
- CM_TPN_NOT_AVAILABLE_NO_RETRY
- CM_TPN_NOT_AVAILABLE_RETRY
- CM_PROGRAM_ERROR_NO_PURGING
- CM_DEALLOCATED_ABEND
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_UNKNOWN_MAP_NAME_REQUESTED
  The supplied map name is unknown to the map routine. No data was sent.
- CM_MAP_ROUTINE_ERROR
  The map routine encountered a problem with the user data. No data was sent.
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  .
  – The conversation is not in **Send**, **Send-Pending**, **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** state.

– The conversation is in **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** state and the program receives a take-commit notification not ending in DATA_OK.
– For a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the conversation's context is in the **Backout Required** condition. The Send_Mapped_Data call is not allowed for this conversation while its context is in this condition.
– The conversation is **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** state, and *send_type* is set to CM_SEND_AND_CONFIRM or CM_SEND_AND_PREP_TO_RECEIVE.
– The conversation is **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** state, the *send_type* is set to CM_SEND_AND_DEALLOCATE, and the *deallocate_type* is not set to CM_DEALLOCATE_ABEND.
– The *send_type* is set to CM_SEND_AND_DEALLOCATE and the following conditions are also true:

  - The *deallocate_type* is set to CM_DEALLOCATE_FLUSH or CM_DEALLOCATE_CONFIRM.

  - The *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the conversation is included in a transaction.

– CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:

  .
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *send_length* exceeds the range permitted by the implementation.
  - The *conversation_type* characteristic is set to:

    CM_BASIC_CONVERSATION

  - The *send_type* is set to CM_SEND_AND_PREP_TO_RECEIVE and the following conditions are also true:
    The *prepare_to_receive_type* is set to:.
    CM_PREP_TO_RECEIVE_SYNC_LEVEL

    • The *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM

    • The conversation is included in a transaction.

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

  - The *send_type* is set to CM_SEND_AND_DEALLOCATE and the following conditions are also true:

    • The *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL

    • The *sync_level* is se to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM

    • The conversation is included in a transaction.

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

• CM_OPERATION_NOT_ACCEPTED

- CM_PRODUCT_SPECIFIC_ERROR

The following values are returned only when *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and the conversation is included in a transaction:

- CM_TAKE_BACKOUT
- CM_DEALLOCATED_ABEND_BO
- CM_RESOURCE_FAIL_NO_RETRY_BO
- CM_RESOURCE_FAILURE_RETRY_BO

The following return codes apply to full-duplex conversations.

- CM_OK
- CM_ALLOCATION_ERROR
- CM_DEALLOCATED_ABEND
- CM_DEALLOCATED_ABEND_SVC
- CM_DEALLOCATED_ABEND_TIMER
- CM_DEALLOCATED_CONFIRM_REJECT
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_DEALLOCATED_NORMAL_ (OSI TP CRM only)
- CM_UNKNOWN_MAP_NAME_REQUESTED
  The supplied map name is unknown to the map routine, no data was sent.
- CM_MAP_ROUTINE_ERROR
  The map routine encountered a problem with the user data, no data was sent.
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The conversation is not in **Send-Receive**, **Send-only**, **Sync-Point**, or **Sync-Point-Deallocate** state.
  - The conversation is basic and in Send-Receive or **Send-Only** state, the *send_type* is set to CM_SEND_AND_DEALLOCATE, the *deallocate_type* is not set to CM_DEALLOCATE_ABEND, and the data does not end on a logical record boundary.
  - The conversation is in **Sync-Point** or **Sync-Point Deallocate** state and the program received a take-commit notification not ending in DATA_OK.
  - The conversation is in **Sync-Point** or **Sync-Point Deallocate** state, the *send_type* is set to CM_SEND_AND_DEALLOCATE, and the *deallocate_type* is not set to CM_DEALLOCATE_ABEND.
  - For a conversation with *sync_level* set to CM_SYNC_POINT_NO_CONFIRM, this return code indicates one of the following:
    - The conversation's context is in the **Backout-Required** condition. The Send_Data call is not allowed for this conversation while its context is in this condition.
    - The local program has received a *status_received* value of **CM_JOIN_TRANSACTION** and must issue a *tx_begin* call to the X/Open TX interface to join the transaction.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.

- The *send_length* exceeds the range permitted by the implementation. The maximum value of the length in each implementation is at least 32767.
- The *send_type* is set to CM_SEND_AND_DEALLOCATE and the following conditions are also true:
- The *deallocate_type* is set to CM_DEALLOCATE_FLUSH or CM_DEALLOCATE_CONFIRM.
- The *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM.
- The conversation is included in a transaction.
- The *send_type* is set to CM_SEND_AND_DEALLOCATE and the following conditions are also true:
- The *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL
- The *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM
- The conversation is included in a transaction.
- The program is not the superior for the conversation.
- The *conversation_type* characteristic is set to CM_BASIC_CONVERSATION

- CM_OPERATION_NOT_ACCEPTED
- CM_OPERATION_INCOMPLETE
- CM_CONVERSATION_CANCELLED
- CM_PRODUCT_SPECIFIC_ERROR

The following values are returned only when *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, the state is Send-Receive and the conversation is currently included in a transaction.

- CM_TAKE_BACKOUT
- CM_DEALLOCATED_ABEND_BO
- CM_RESOURCE_FAIL_NO_RETRY_BO
- CM_RESOURCE_FAILURE_RETRY_BO
- CM_CONV_DEALLOC_AFTER_SYNCPT
- CM_INCLUDE_PARTNER_REJECT_BO

State Changes

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

- The conversation enters Receive state when Send_data is issued with *send type* set to CM_SEND_AND_PREP_TO_RECEIVE and any of the following conditions are true.
  - *Prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_FLUSH
  - *Prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_CONFIRM
  - *Prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* is set to CM_NONE or CM_CONFIRM
  - *Prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, but the conversation is not currently included in a transaction.
- The conversation enters **Defer-Receive** state when Send_Data is issued with *send_type* set to CM_SEND_AND_PREP_TO_RECEIVE, *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and the conversation is included in a transaction.
- The conversation enters **Reset** state when Send_Data is issued with *send_type* set to CM_SEND_AND_DEALLOCATE and any of the following conditions is true:

- – *Deallocate_type* is set to CM_DEALLOCATE_ABEND
- – *Deallocate_type* is set to CM_DEALLOCATE_FLUSH
- – *Deallocate_type* is set to CM_DEALLOCATE_CONFIRM
- – *Deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is set to CM_NONE or CM_CONFIRM
- – *Deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, but the conversation is not currently included in a transaction.

- The conversation enters **Defer-Deallocate** state when Send_Data is issued with *send type* set to CM_SEND_AND_DEALLOCATE, *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and
- The conversation enters **Send** state when Send_Data is issued in Send-Pending state with *send_type* set to CM_BUFFER_DATA CM_SEND_AND_FLUSH, or CM_SEND_AND_CONFIRM.
- No state change occurs when Send_Data is issued in Send state with *send_type* set to CM_BUFFER_DATA_CM_SEND_AND_FLUSH, or CM_SEND_AND_CONFIRM.

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

- The conversation enters **Receive-Only** state when the Send_Data call is issued in **Send-Receive** state with *send_type* set to CM_SEND_AND_DEALLOCATE, and one of the following conditions is true:

  – *Deallocate_type* is set to CM_DEALLOCATE_FLUSH

  – *Deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is set to CM NONE

  – *Deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, and the conversation is not currently included in a transaction.

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

  – The call is issued in **Send-Only** state.

  – The call is issued in **Send-Receive**, **Send-Only**, **Sync-Point**, or **Sync-Point-Deallocate** state and *deallocate_type* is set to CM_DEALLOCATE_ABEND

- The conversation enters **Defer-Deallocate** state when the Send_Data call is issued with *send_type* set to CM_SEND_DEALLOCATE, *sync_level* set to CM_SYNC_POINT_NO_CONFIRM, and the conversation included in a transaction.

- No state change occurs when Send_Data is issued in **Send-Receive** or **Send-Only** state with *send_type* set to CM_BUFFER_DATA or CM_SEND_AND_FLUSH.

Usage Notes

1. The local system may buffer data to be sent to the remote system unit until it accumulates a sufficient amount of data for transmission (from one or more Send_Mapped_Data calls), or until the local program issues a call that causes the system to flush its send buffer. The map routine is responsible for any

buffering and determines when data should be sent. Therefore, a flush call may or may not have an effect.

2. For a half-duplex conversation, when *control_information_received* indicates CM_REQ_TO_SEND_RECEIVED, CM_RTS_RCVD_AND_EXP_DATA_AVAIL, or CM_RTS_RCVD_AND_EXP_MAP_DATA_AVAIL or the remote program is requesting the local program's end of conversation to enter Receive state, which places the remote program's end of the conversation in **Send** state. See “Request_To_Send (CMRTS)” on page 246 for a discussion of how a program can place its end of a conversation in Receive state.

3. When issued during a mapped conversation, the Send_Mapped_Data call sends one complete data record. The data record consists entirely of data. CPI-C does not examine the data for logical record length fields. It is this specification of a complete data record, at send time by the local program and what it sends, that is indicated to the remote program by the *data_received* parameter of the Receive_Mapped_Data call.

   For example, consider a mapped conversation where the local program issues tow Send_Data calls with *send_length* set to 30 and 50 respectively. (These numbers are simple for explanatory purposes.) The local program then issues Flush and the 80 bytes of data are sent to the remote system. The remote program now issues Receive_Mapped_Data with *requested length* set to a sufficiently large value of 1000. The remote program will receive back only 30 bytes of data, indicated by the *received_length* parameter, because this is a complete data record. The completeness of the data record is indicated by the *data_received* variable, which will be set to CM_COMPLETE_DATA_RECEIVED.

   The remote program receives the remaining 50 bytes of data (from the second Send_Mapped_Data) when it performs a second Receive_Mapped_Data with *requested_length* set to a value greater than or equal to 50.

   Note: The *received_length* may be different from the *send_length* because of the differences in how data is represented locally in each partner system.

4. If the map routine is unable to process the user data because an incorrect map_name was identified or the map routine encountered an error during processing, the *return_code* is set appropriately. Nothing is sent and the conversation is not aborted. Control is given back to the program for appropriate action.

5. The *send_length* is local information for the map routine. The *send_length* value that is used by the CRM is calculated by the map routine after all transformations are completed.

6. Send_Data is often used in conjunction with other call, such as Flush, Confirm, and Prepare_To_Receive. Contrast this usage with the equivalent function available from the use of the Set_Send_Type call prior to issuing a call to Send_Data.

7. A program must not specify a value in the *send_length* parameter that is greater than the maximum the implementation can support. The maximum may vary from system-to-system. The maximum also depends on the *map_name* parameter.

   Every CPI-C product must be able to send buffers with a length of 32767 bytes. Map routines written should describe the length of the data buffer before encoding, if the encoded buffer has a length of 32767 bytes.

   The application program should not use a *send_length* larger than this.

8. When *control_information_received* indicates that expedited data is available to be received, subsequent calls with this parameter will continue to indicate that expedited data is available until the expedited data has been received by the program.

Related Information

"Conversation Types" on page 19 provides more information on mapped and basic conversations.

"Data Buffering and Transmission" on page 44 provides information about data transmission control.

Chapter 3, "Program-to-Program Communication Example Flows" on page 67 show programs using the Send_Data call.

For a mapped conversation with a non-null map name, the largest amount of data a program can send in a Send_Mapped_Data call and the largest amount of data that can be received by the program in a Receive_Mapped_Data call depends on the map name.

"Receive_Mapped_Data (CMRCVM)" on page 231 provides more information about the *data_received* parameter.

"Set_Send_Type (CMSST)" on page 351 provides more information about the *send_type* conversation characteristic and the use of it in combination with calls to Send_Data.

The *SNA Transaction Programmers Reference Manual for LU Type 6.2* provides more information about mapped conversations.

Set_AE_Qualifier (CMSAEQ)

OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32

Set_AE_Qualifier (CMSAEQ) is used by a program to set the *AE_qualifier*, *AE_qualifier_length*, and *AE_qualifier_format* characteristics for a conversation. Set_AE_Qualifier overrides the current values that were originally acquired from the side information using *sym_dest_name*.

Issuing this call does not change the information in the side information. It only changes the *AE_qualifier*, the *AE_qualifier_length*, and the *AE_qualifier_format* characteristics for this conversation.

The *AE_qualifier* conversation characteristic identifies the OSI application-entity qualifier associated with the remote program. It may be set by the conversation initiator to identify the location of the remote program.

Notes:

1. A program cannot issue Set_AE_Qualifier after an Allocate call is issued. Only the program that initiated the conversation (issued the Initialize_Conversation call) can issue Set_AE_Qualifier.
2. The *AE_qualifier* characteristic is used only by an OSI TP CRM.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

AE_qualifier *(input)*
Specifies the application-entity-qualifier that distinguishes the application-entity at the application-process where the remote program is located.

AE_qualifier_length *(input)*
Specifies the length of *AE_qualifier*. The length can be from 1 to 1024 bytes.

AE_qualifier_format *(input)*

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

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

return_code *(output)*

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *AE_qualifier_length* is set to a value less than 1 or greater than 1024.
  - The *AE_qualifier_format* specifies an undefined value.
  - The *partner_ID* characteristic is set to a non-null value.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is not in **Initialize** state.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

1. Specify *AE_qualifier* using the local system's native encoding. CPI Communications automatically converts the *AE_qualifier* from the native encoding where necessary.
2. If a *return_code* other than CM_OK is returned on the call, the *AE_qualifier*, the *AE_qualifier_length*, and the *AE_qualifier_format* conversation characteristics remain unchanged.
3. The *AE_qualifier* may be either a distinguished name or an integer represented as a sequence of decimal digits. Distinguished names can have any format and syntax that can be recognized by the local system.

Related Information

"Side Information" on page 23 and note 4 of Table 61 on page 650 provide further discussion of the *AE_qualifier* conversation characteristic.

"Automatic Conversion of Characteristics" on page 41 provides further information on the automatic conversion of the *AE_qualifier* parameter.

Set_Allocate_Confirm (CMSAC)

OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32

Set_Allocate_Confirm (CMSAC) is used by a program to set the *allocate_confirm* characteristic for a given conversation. Set_Allocate_Confirm overrides the value that was assigned when the Initialize_Conversation call was issued.

The conversation initiator uses the Set_Allocate_Confirm call to indicate whether or not a positive confirmation is required when the remote program has accepted the conversation.

Notes:

1. A program cannot issue Set_Allocate_Confirm after an Allocate call is issued. Only the program that initiates the conversation (issues the Initialize_Conversation call) can issue Set_Allocate_Confirm.
2. The *allocate_confirm* characteristic is used only by an OSI TP CRM.

Format

```
CALL CMSAC(conversation_ID,
           allocate_confirm,
           return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

allocate_confirm *(input)*
Specifies whether the program is to receive notification when the remote program confirms its acceptance of the conversation. The *allocate_confirm* variable can have one of the following values:

- CM_ALLOCATE_NO_CONFIRM
  Specifies that the program is not to receive notification when the remote program confirms its acceptance of the conversation.
- CM_ALLOCATE_CONFIRM
  Specifies that the program is to receive notification when the remote program confirms its acceptance of the conversation.

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

- CM_OK
- CM_CALL_NOT_SUPPORTED

- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  – The *conversation_ID* specifies an unassigned identifier.
  – The *allocate_confirm* specifies an undefined value.
  – The *allocate_confirm* specifies CM_ALLOCATE_CONFIRM, and the conversation is using an LU 6.2 CRM.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is not in **Initialize** state.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

1. If a *return_code* other than CM_OK is returned on the call, the *allocate_confirm* conversation characteristic remains unchanged.
2. When the remote program confirms its acceptance of the conversation, the initiating program is notified by receiving a *control_information_received* value of CM_ALLOCATE_CONFIRMED or CM_ALLOCATE_CONFIRMED_WITH_DATA on a subsequent call.
3. After the remote program has accepted the conversation by issuing an Accept_Conversation or Accept_Incoming call, it confirms the acceptance by issuing any call other than a Cancel_Conversation, Deallocate with *deallocate_type* of CM_DEALLOCATE_ABEND, or Set_* or Extract_* call. The remote program rejects the conversation by issuing a Cancel_Conversation call or a Deallocate call with *deallocate_type* of CM_DEALLOCATE_ABEND as its first operation on the conversation (other than a Set_* or Extract_* call).
4. The program that initiates the conversation (issues Initialize_Conversation) must set *allocate_confirm* to CM_ALLOCATE_CONFIRM if it is expecting *initialization_data* to be returned from the remote program after the remote program confirms its acceptance of the conversation.

Set_AP_Title (CMSAPT)

OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32

Set_AP_Title (CMSAPT) is used by a program to set the *AP_title*, *AP_title_length*, and *AP_title_format* characteristics for a conversation. Set_AP_Title overrides the current values that were originally acquired from the side information using *sym_dest_name*.

Issuing this call does not change the values in the side information. It only changes the *AP_title*, the *AP_title_length*, and the *AP_title_format* characteristics for this conversation.

The *AP_title* conversation characteristic identifies the OSI application-process title associated with the remote program. It may be set by the conversation initiator to identify the location of the remote program.

Notes:

1. A program cannot issue Set_AP_Title after an Allocate call is issued. Only the program that initiated the conversation (issued the Initialize_Conversation call) can issue Set_AP_Title.
2. The *AP_title* characteristic is used only by an OSI TP CRM.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

AP_title *(input)*
Specifies the title of the application-process where the remote program is located.

AP_title_length *(input)*
Specifies the length of *AP_title*. The length can be from 1 to 1024 bytes.

AP_title_format *(input)*
Specifies the format of *AP_title*. The *AP_title_format* variable can have one of the following values:

- CM_DN
  Specifies that the *AP_title* is a distinguished name.
- CM_OID
  Specifies that the *AP_title* is an object identifier.

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  – The *conversation_ID* specifies an unassigned identifier.
  – The *AP_title_length* is set to a value less than 1 or greater than 1024.
  – The *AP_title_format* specifies an undefined value.
  – The *partner_ID* characteristic is set to a non-null value.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is not in **Initialize** state.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

1. Specify *AP_title* using the local system's native encoding. CPI Communications automatically converts the *AP_title* from the native encoding where necessary.
2. If a *return_code* other than CM_OK is returned on the call, the *AP_title*, the *AP_title_length*, and the *AP_title_format* conversation characteristics remain unchanged.
3. The *AP_title* may be either a distinguished name or an object identifier. Distinguished names can have any format and syntax that can be recognized by the local system. Object identifiers are represented as a series of digits separated by periods (for example, *n.nn.n.nnn*).

Related Information

“Side Information” on page 23 and note 4 of Table 61 on page 650 provide further discussion of the *AP_title* conversation characteristic.

“Automatic Conversion of Characteristics” on page 41 provides more information on the automatic conversion of the *AP_title* parameter.

Set_Application_Context_Name (CMSACN)

OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32

Set_Application_Context_Name (CMSACN) is used by a program to set the *application_context_name* and *application_context_name_length* characteristics for a conversation. Set_Application_Context_Name overrides the current values that were originally acquired from the side information using *sym_dest_name*.

The *application_context_name* conversation characteristic identifies the OSI application-context used on the conversation. It may be set by the conversation initiator.

Issuing this call does not change the values in the side information. It only changes the *application_context_name* and *application_context_name_length* characteristics for this conversation.

Note: The *application_context_name* characteristic is used only by an OSI TP CRM.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

application_context_name *(input)*
Specifies the name of the application context to be used on the conversation. The length can be 1-256 bytes.

application_context_name_length *(input)*
Specifies the length of the application context name to be used on the conversation startup request.

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is not in the **Initialize** state.

- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned identifier.
  - The *application_context_name_length* is set to a value less than 1 or greater than 256.
  - The *partner_ID* characteristic is set to a non-null value.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

1. Specify *application_context_name* using the local system's native encoding. CPI Communications automatically converts the *application_context_name* from the native encoding where necessary.
2. If a *return_code* other than CM_OK is returned on the call, the *application_context_name* and the *application_context_name_length* conversation characteristics remain unchanged.
3. The application context name is an object identifier and is represented as a series of digits separated by periods. For example, the application context defined by the Open System Environment Implementers' Workshop (OIW) for UDT with Commit Profiles is represented as "1.3.14.15.5.1.0" and the application context for Application Supported Transactions using UDT is represented as "1.3.14.15.5.2.0".

Related Information

"Side Information" on page 23 provides more information on the *application_context_name* conversation characteristic.

"Automatic Conversion of Characteristics" on page 41 provides further information on the automatic conversion of the *application_context_name* parameter.

Set_Begin_Transaction (CMSBT)

OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32

Set_Begin_Transaction (CMSBT) is used by a program to set the *begin_transaction* characteristic for a given conversation. Set_Begin_Transaction overrides the value that was assigned when the Initialize_Conversation call was issued.

The superior program uses the Set_Begin_Transaction call to indicate whether the subordinate is to be explicitly included in the current transaction or implicitly included, if and when any conversation activity occurs.

Note: The *begin_transaction* characteristic is used only by an OSI TP CRM.

Format

```
CALL CMSBT(conversation_ID,
           begin_transaction,
           return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

begin_transaction *(input)*
Specifies whether the superior, when a new transaction is started, will explicitly or implicitly ask that the subordinate program join the transaction. The *begin_transaction* variable can have one of the following values:

- CM_BEGIN_IMPLICIT
  Specifies that the superior implicitly asks that the subordinate join the transaction by issuing one of the following calls from **Initialize**, **Send**, **Send-Pending**, or **Send-Receive** states.
  CMALLC — Allocate
  CMCFM — Confirm
  CMINCL — Include_Partner_In_Transaction
  CMPREP — Prepare
  CMPTR — Prepare_To_Receive
  CMRCV — Receive
  CMSEND — Send_Data
  CMSERR — Send_Error
- CM_BEGIN_EXPLICIT
  Specifies that the superior explicitly asks that the subordinate join the transaction by use of the Include_Partner_In_Transaction (CMINCL) call.

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_STATE_CHECK
This value indicates that the conversation is in **Initialize-Incoming** state.
- CM_PROGRAM_PARAMETER_CHECK
This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned identifier.
  - The *begin_transaction* specifies an undefined value.
  - The *transaction_control* is set to CM_CHAINED_TRANSACTIONS.
  - The program is not the superior for the conversation.
- CM_OPERATION_NOT_ACCEPTED
This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

1. If a *return_code* other than CM_OK is returned on the call, the *begin_transaction* conversation characteristic remains unchanged.
2. This call does not apply to any previous call operation still in progress.
3. The remote program receives the request to join the transaction as a *status_received* indicator of CM_JOIN_TRANSACTION on a Receive call it issues.
4. If the superior is not in transaction when it issues an Allocate, Confirm, Include_Partner_In_Transaction, Prepare, Prepare_To_Receive, Receive, Send_Data, or Send_Error call, the *begin_transaction* characteristic is ignored, and the subordinate is not asked to join a transaction.
5. If *begin_transaction* is set to CM_BEGIN_IMPLICIT, the subordinate is asked to join the transaction only when the Allocate, Confirm, Include_Partner_In_Transaction, Prepare, Prepare_To_Receive, Receive, Send_Data or Send_Error call is returned with a *return_code* of CM_OK.
6. The call is not associated with any conversation queue. When a conversation uses queue-level non-blocking, the call does not return CM_OPERATION_NOT_ACCEPTED on the conversation.

Related Information

"Chained and Unchained Transactions" on page 61 discusses chained and unchained transactions.

"Joining a Transaction" on page 61 discusses how a program requests the partner program to join a transaction.

Set_Confirmation_Urgency (CMSCU)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32

Set_Confirmation_Urgency (CMSCU) is used by a program to set the *confirm_urgency* characteristic for a given conversation. Set_Confirmation_Urgency overrides the value that was assigned when the Initialize_Conversation, Accept_Conversation, or Initialize_For_Incoming call was issued.

Programs would use the Set_Confirmation_Urgency call to indicate whether to optimize the Prepare_to_Receive flows for reduced link traffic or immediate response.

Note: The *confirmation_urgency* characteristic is used only for a half-duplex conversation.

Format

```
CALL CMSCU(conversation_ID,
           confirmation_urgency,
           return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

confirmation_urgency *(input)*
Specifies whether the response to a Prepare_To_Receive call that requests confirmation will be sent immediately. The *confirmation_urgency* variable can have one of the following values:

- CM_CONFIRMATION_NOT_URGENT
  Specifies that the remote program's response to the confirmation request may not be sent immediately.
- CM_CONFIRMATION_URGENT
  Specifies that the remote program's response to the confirmation request will be sent immediately.

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

- CM_OK
- CM_CALL_NOT_SUPPORTED

- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned identifier.
  - The *confirmation_urgency* variable specifies an undefined value.
  - The *send_receive_mode* of the conversation is CM_FULL_DUPLEX.
  - The *sync_level* is set to CM_NONE or CM_SYNC_POINT_NO_CONFIRM.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

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

2. When the local program issues a Prepare_To_Receive call with *prepare_to_receive_type* set to CM_PREPARE_TO_RECEIVE_CONFIRM or CM_PREPARE_TO_RECEIVE_SYNC_LEVEL with *sync_level* set to CM_CONFIRM, the remote CRM may optimize link usage by buffering the response generated by the Confirmed call until the remote program issues another call. This may increase the time the local program must wait for control to be returned to it.

   By issuing a Set_Confirmation_Urgency call with the *confirmation_urgency* parameter set to CM_CONFIRMATION_URGENT before issuing the Prepare_To_Receive call, the local program can request that the response from the Confirmed call be sent to the local program as soon as it is issued by the remote program.

Related Information

See “Prepare_To_Receive (CMPTR)” on page 208 for information on requesting confirmation.

Set_Conversation_Security_Password (CMSCSP)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X*				X	X*	X	X*	X

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

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

The *security_password* conversation characteristic contains the password associated with the user identification to be used on the conversation. It may be set by the conversation initiator.

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

X* AIX prior to Version 3 Release 1, OS/2 prior to Communications Server, and VM support this function in a product-specific extension call.

- For AIX, see “Set_Conversation_Security_Password (XCSCSP)” on page 399.
- For OS/2, see “Set_Conversation_Security_Password (XCSCSP)” on page 615.
- For VM, see “Set_Conversation_Security_Password (XCSCSP)” on page 562.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

security_password *(input)*
Specifies the password to be included in the conversation startup request. The partner system uses this value and the user ID to validate the user's access to

the remote program. The password is stored temporarily by node services, and is erased at the successful completion of an Allocate call.

security_password_length *(input)*
Specifies the length of the password. The length can be from 0 to 10 bytes. If zero, the *security_password_length* characteristic is set to zero (effectively setting the *security_password* characteristic to the null string), and the *security_password* parameter on this call is ignored.

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The conversation is not in **Initialize** state.
  - *conversation_security_type* is not set to CM_SECURITY_PROGRAM or CM_SECURITY_PROGRAM_STRONG.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *security_password_length* is less than 0 or greater than 10.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

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

Related Information

"Automatic Conversion of Characteristics" on page 41 provides further information on the automatic conversion of the *security_password* parameter.

"Conversation Security" on page 51 provides further information on security.

"Set_Conversation_Security_Type (CMSCST)" on page 295 provides more information on the *conversation_security_type* characteristic.

"Set_Conversation_Security_User_ID (CMSCSU)" on page 298 provides more information on the *security_user_ID* characteristic.

Set_Conversation_Security_Type (CMSCST)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X*				X	X*	X	X*	X

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

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

The conversation initiator uses the Set_Conversation_Security_Type call to indicate the type of authorization to be used during conversation initialization.

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

X* AIX prior to Version 3 Release 1, OS/2 prior to Communications Server, and VM support this function in a product-specific extension call.

- For AIX, see “Set_Conversation_Security_Type (XCSCST)” on page 401.
- For OS/2, see “Set_Conversation_Security_Type (XCSCST)” on page 616.
- For VM, see “Set_Conversation_Security_Type (XCSCST)” on page 564.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

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

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

- CM_SECURITY_NONE
  No access security information is included in the conversation startup request.

- CM_SECURITY_SAME
  The security parameters maintained by node services for the program's current context when the program issues the Allocate call are used to set the access security information included in the conversation startup request.
- CM_SECURITY_PROGRAM
  The values of the *security_user_ID* and *security_password* characteristics are used to set the access security information included in the conversation startup request.
- CM_SECURITY_PROGRAM_STRONG
  The values of the *security_user_ID* and *security_password* characteristics are used to set the access security information included in the conversation startup request. The local CRM ensures that the *security_password* is not exposed in clear-text form on the physical network. If the local CRM cannot ensure this, then the subsequent Allocate request will fail with a *return_code* of CM_SECURITY_NOT_SUPPORTED.

  Currently, OS/400 is the only product/platform that implements CM_SECURITY_PROGRAM_STRONG.

- CM_SECURITY_DISTRIBUTED
  The security parameters for the program's current context are used to generate authentication tokens using a distributed security service. The tokens are used as the access security information included in the conversation startup request. If the remote system does not accept the type of tokens generated by the local system, then the request fails locally. A locally failed request does not send any data to the remote system.

  Currently, no product/platform implements CM_SECURITY_DISTRIBUTED.

- CM_SECURITY_MUTUAL
  The security parameters for the program's current context are used to generate authentication tokens using a distributed security service. The tokens are used as the access security information included in the conversation startup request. Furthermore, the local CRM is requested to authenticate the partner principal before any user information is transmitted to the remote CRM.

  Currently, no product/platform implements CM_SECURITY_MUTUAL.

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is not in **Initialize** state.
- CM_PARM_VALUE_NOT_SUPPORTED
  This value indicates that the *conversation_security_type* specifies CM_SECURITY_PROGRAM, CM_SECURITY_PROGRAM_STRONG, CM_SECURITY_DISTRIBUTED, or CM_SECURITY_MUTUAL and the value is not supported by the local system.

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

State Changes

This call does not cause any state changes.

Usage Notes

1. A *conversation_security_type* of CM_SECURITY_SAME is intended for use between nodes which have the same set of user IDs and which accept user validation performed on one node as verifying the user for all nodes. A password is not used in this case.
2. A *conversation_security_type* of CM_SECURITY_PROGRAM or CM_SECURITY_PROGRAM_STRONG requires that a user ID and password be supplied for inclusion in the conversation startup request. These may come from side information or be set by the program using the Set_Conversation_Security_User_ID and Set_Conversation_Security_Password calls.
3. If a *return_code* other than CM_OK is returned on the call, the *conversation_security_type* is unchanged.
4. A *conversation_security_type* of CM_SECURITY_MUTUAL is intended for usage when the local program wants the local CRM to authenticate that the remote program or CRM is actually the partner principal specified in the program binding.
5. Certain combinations of *conversation_security_type* value and required_user_name_type value (from the program binding) causes the local CRM to reject an Allocate request with a *return_code* value of CM_SECURITY_NOT_SUPPORTED. The incompatible combinations are shown in Table 8 on page 52.
6. For an OSI TP CRM, the type of access security information sent for CM_SECURITY_SAME is implementation defined.

Related Information

"Conversation Security" on page 51 provides further information on security.

"Set_Conversation_Security_Password (CMSCSP)" on page 292 discusses setting the *security_password* characteristic.

"Set_Conversation_Security_User_ID (CMSCSU)" on page 298 discusses setting the *security_user_ID* characteristic.

"Set_Partner_ID (CMSPID)" on page 323 discusses setting the *partner_ID* characteristic.

"Program Binding" on page 658 provides further information on the program binding.

Set_Conversation_Security_User_ID (CMSCSU)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X*				X	X*	X	X*	X

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

The *security_user_id* conversation characteristic contains the user identification associated with the conversation. It may be set by the conversation initiator.

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

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

X* AIX prior to Version 3 Release 1, OS/2 prior to Communications Server, and VM support this function in a product-specific extension call.

- For AIX, see “Set_Conversation_Security_User_ID (XCSCSU)” on page 403.
- For OS/2, see “Set_Conversation_Security_User_ID (XCSCSU)” on page 617.
- For VM, see “Set_Conversation_Security_User_ID (XCSCSU)” on page 566.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

security_user_ID *(input)*
Specifies the user ID to be included in the conversation startup request. The partner system uses this value and the password to validate the user's access

to the remote program. In addition, the partner system may use the user ID for auditing or accounting purposes.

security_user_ID_length *(input)*
Specifies the length of the user ID. The length can be from 0 to 10 bytes. If zero, the *security_password_length* characteristic is set to zero (effectively setting the *security_password* characteristic to the null string), and the *security_password* parameter on this call is ignored.

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The conversation is not in **Initialize** state.
  - The *conversation_security_type* is not set to CM_SECURITY_PROGRAM or CM_SECURITY_PROGRAM_STRONG.
- CM_PROGRAM_PARAMETER_CHECK
  This return code indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *security_user_ID_length* is less than 0 or greater than 10.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

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

Related Information

“Automatic Conversion of Characteristics” on page 41 provides further information on the automatic conversion of the *security_user_ID* parameter.

“Conversation Security” on page 51 provides further information on security.

"Set_Conversation_Security_Password (CMSCSP)" on page 292 provides more information on the *security_password* characteristic.

"Set_Conversation_Security_Type (CMSCST)" on page 295 provides more information on the *conversation_security_type* characteristic.

Set_Conversation_Type (CMSCT)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

Set_Conversation_Type (CMSCT) is used by a program to set the *conversation_type* characteristic for a given conversation. It overrides the value that was assigned when the Initialize_Conversation or Initialize_For_Incoming call was issued.

The *conversation_type* conversation characteristic indicates whether the data exchange on the conversation will follow basic or mapped rules.

Notes:

1. A program cannot use Set_Conversation_Type after an Allocate has been issued. Only the program that initiates the conversation (using the Initialize_Conversation call) can issue the Set_Conversation_Type call.
2. When using an LU 6.2 CRM, only the program that initiates the conversation (using the Initialize_Conversation call) can issue the Set_Conversation_Type call, and it must be issued before the Allocate is issued. When using an OSI TP CRM, because the mapped/basic indication is not carried by the protocol, the recipient of the conversation request must use Initialize_For_Incoming and Set_Conversation_Type to override the *conversation_type* default of CM_MAPPED_CONVERSATION. The recipient must issue these calls before the Accept_Incoming call is issued.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

conversation_type *(input)*
Specifies the type of conversation to be allocated when Allocate is issued. The *conversation_type* variable can have one of the following values:

- CM_BASIC_CONVERSATION
  Specifies the allocation of a basic conversation.
- CM_MAPPED_CONVERSATION
  Specifies the allocation of a mapped conversation.

return_code *(output)*

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

- CM_OK
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is not in **Initialize** or **Initialize-Incoming** state.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *conversation_type* specifies an undefined value.
  - The *conversation_type* is set to CM_MAPPED_CONVERSATION, but *fill* is set to CM_FILL_BUFFER.
  - The *conversation_type* is set to CM_MAPPED_CONVERSATION, but a prior call to Set_Log_Data is still in effect.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. Because of the detailed manipulation of the data and resulting complexity of error conditions, the use of basic conversations should be regarded as an advanced programming technique.
2. If a *return_code* other than CM_OK is returned on the call, the *conversation_type* conversation characteristic is unchanged.

Related Information

"Conversation Types" on page 19 and the "Usage Notes" section of "Send_Data (CMSEND)" on page 249 provide more information on the differences between mapped and basic conversations.

Set_Deallocate_Type (CMSDT)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

Set_Deallocate_Type (CMSDT) is used by a program to set the *deallocate_type* characteristic for a given conversation. Set_Deallocate_Type overrides the value that was assigned when the Initialize_Conversation, Accept_Conversation, or Initialize_For_Incoming call was issued.

The Set_Deallocate_Type call is used by programs to identify the processing option to be used when the local program issues a Deallocate call.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

deallocate_type *(input)*
Specifies the type of deallocation to be performed. The *deallocate_type* variable can have one of the following values:

- CM_DEALLOCATE_SYNC_LEVEL
  Perform deallocation based on the *sync_level* characteristic in effect for this conversation:
  – If *sync_level* is set to CM_NONE, or if *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM but the conversation is not currently included in a transaction, execute the function of the Flush call and deallocate the conversation normally.
  – For half-duplex conversations, if *sync_level* is set to CM_CONFIRM, or if *sync_level* is set to CM_SYNC_POINT but the conversation is not currently included in a transaction, execute the function of the Confirm call. If the Confirm call is successful (as indicated by a return code of CM_OK on the Deallocate call or a return code of CM_OK on the Send_Data call with *send_type* set to CM_SEND_AND_DEALLOCATE), deallocate the conversation normally. If the Confirm call is not successful, the state of the conversation is determined by the return code.
  – If *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and the conversation is included in a transaction, defer the deallocation until the program issues a resource recovery commit call. If the commit call is successful, the conversation is deallocated normally. If the commit is not successful or if the

program issues a resource recovery backout call instead of a commit, the conversation is not deallocated. See "Deallocate (CMDEAL)" on page 143 for more information about deallocating conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.

- CM_DEALLOCATE_FLUSH
  Execute the function of the Flush call and deallocate the conversation normally.
- CM_DEALLOCATE_CONFIRM
  Execute the function of the Confirm call. If the Confirm call is successful (as indicated by a return code of CM_OK on the Deallocate call or a return code of CM_OK on the Send_Data call with *send_type* set to CM_SEND_AND_DEALLOCATE), deallocate the conversation normally. If the Confirm call is not successful, the state of the conversation is determined by the return code.
- CM_DEALLOCATE_ABEND
  For half-duplex conversations, execute the function of the Flush call when the conversation is in **Send** state and deallocate the conversation abnormally. Data purging can occur when the conversation is in **Receive** state. If the conversation is a basic conversation, logical-record truncation can occur when the conversation is in **Send** state.

  For full-duplex conversations, execute the function of the Flush call when the conversation is in **Send-Receive** or **Send-Only** state and deallocate the conversation abnormally. Data purging can occur when the conversation is in **Send-Receive** or **Receive-Only** state. If the conversation is basic, logical-record truncation can occur when the conversation is in **Send-Receive** or **Send-Only** state.

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

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  – The *conversation_ID* specifies an unassigned conversation identifier.
  – The *deallocate_type* specifies an undefined value.
  – The *deallocate_type* is set to CM_DEALLOCATE_FLUSH, *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and *transaction_control* is set to CM_CHAINED_TRANSACTIONS.
  – The conversation is using an LU 6.2 CRM, *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, and *sync_level* is set to CM_NONE, CM_SYNC_POINT, or CM_SYNC_POINT_NO_CONFIRM.
  – The converation is using an OSI TP CRM, *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and *transaction_control* is set to CM_CHAINED_TRANSACTIONS.
- CM_PROGRAM_STATE_CHECK
  The conversation is in **Initialize-Incoming** state.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. A *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL is used by a program to deallocate a conversation based on the conversation's synchronization level.
   - For half-duplex conversations:
     - If *sync_level* is set to CM_NONE, or if *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM but the conversation is not currently included in a transaction, the conversation is unconditionally deallocated.
     - If *sync_level* is set to CM_CONFIRM, or if *sync_level* is set to CM_SYNC_POINT but the conversation is not currently included in a transaction, the conversation is deallocated when the remote program responds to the confirmation request by issuing the Confirmed call. The conversation remains allocated when the remote program responds to the confirmation request by issuing the Send_Error call.
     - If *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and the conversation is included in a transaction, the deallocation is deferred until the program issues a resource recovery commit call. If the commit call is successful, the conversation is deallocated normally. If the commit is not successful or if the program issues a resource recovery backout call instead of a commit, the conversation is not deallocated.
   - For full-duplex conversations:
     - If *sync_level* is set to CM_NONE, or if *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM but the conversation is not currently included in a transaction, and the Deallocate call is issued in **Send-Receive** state, the program can no longer issue calls associated with the Send queue, but it can continue to issue calls associated with the other conversation queues. If the Deallocate call is issued in **Send-Only** state, the conversation is deallocated.
     - If *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM and the conversation is included in a transaction, the deallocation is deferred until the program issues a resource recovery commit call. If the commit call is successful, the conversation is deallocated normally. If the commit is not successful or if the program issues a backout call instead of a commit, the conversation is not deallocated.
2. A *deallocate_type* set to CM_DEALLOCATE_FLUSH is used by a program to unconditionally deallocate the conversation. This *deallocate_type* value can be used for conversations with *sync_level* set to CM_NONE or CM_CONFIRM. If the conversation is using an OSI TP CRM, it can also be used for conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM if the conversation is not currently included in a transaction. The *deallocate_type* set to CM_DEALLOCATE_FLUSH is functionally equivalent to *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL combined with a *sync_level* set to CM_NONE.

   For a half-duplex conversation, the conversation is deallocated. For a full-duplex conversation, the program can no longer issue calls associated with the Send queue, and the conversation is deallocated if the Deallocate call is

issued in **Send-Only** state. This *deallocate_type* value can be used for conversations with *sync_level* set to CM_NONE.

3. A *deallocate_type* set to CM_DEALLOCATE_CONFIRM is used by a program to conditionally deallocate the conversation, depending on the remote program's response, when the *sync_level* is set to CM_CONFIRM. The *deallocate_type* set to CM_DEALLOCATE_CONFIRM is functionally equivalent to *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL combined with a *sync_level* set to CM_CONFIRM.

   The conversation is deallocated when the remote program responds to the confirmation request by issuing Confirmed. The conversation remains allocated when the remote program responds to the confirmation request by issuing Send_Error.

4. A *deallocate_type* set to CM_DEALLOCATE_ABEND is used by a program to unconditionally deallocate a conversation regardless of its synchronization level and its current state. Specifically, this *deallocate_type* value is used when the program detects an error condition that prevents further useful communications (communications that would lead to successful completion of the transaction).

5. If a *return_code* other than CM_OK is returned on the call, the *deallocate_type* conversation characteristic is unchanged.

Related Information

"Deallocate (CMDEAL)" on page 143 provides further discussion on the use of the *deallocate_type* characteristic in the deallocation of a conversation.

"Set_Sync_Level (CMSSL)" on page 354 provides information on how the *sync_level* characteristic is used in combination with the *deallocate_type* characteristic in the deallocation of a conversation.

Set_Error_Direction (CMSED)

LU 6.2

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

Set_Error_Direction (CMSED) is used by a program to set the *error_direction* characteristic for a given conversation. Set_Error_Direction overrides the value that was assigned when the Initialize_Conversation, the Accept_Conversation, or the Initialize_For_Incoming call was issued.

A program uses the Set_Error_Direction call, when using an LU 6.2 CRM, to indicate whether the error was detected while receiving the data or after receiving the data.

Note: The *error_direction* characteristic is used by an LU 6.2 CRM and only for a half-duplex conversation.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

error_direction *(input)*
Specifies the direction of the data flow in which the program detected an error. This parameter is significant only if Send_Error is issued in **Send-Pending** state (that is, immediately after a Receive on which both data and a conversation status of CM_SEND_RECEIVED are received). Otherwise, the *error_direction* value is ignored when the program issues Send_Error.

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

- CM_RECEIVE_ERROR
  Specifies that the program detected an error in the data it received from the remote program.
- CM_SEND_ERROR
  Specifies that the program detected an error while preparing to send data to the remote program.

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

- CM_OK
- CM_CALL_NOT_SUPPORTED

- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  – The *conversation_ID* specifies an unassigned conversation identifier.
  – The *error_direction* specifies CM_SEND_ERROR and the conversation is using an OSI TP CRM.
  – The *error_direction* specifies an undefined value.
  – The *send_receive_mode* of the conversation is CM_FULL_DUPLEX.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. The *error_direction* conversation characteristic is significant only if Send_Error is issued immediately after a Receive on which both data and a conversation status of CM_SEND_RECEIVED are received (when the conversation is in **Send-Pending** state). In this situation, the Send_Error may result from one of the following errors:

   - An error in the received data (in the receive flow)

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

   Because the system in this situation cannot tell which error occurred, the program has to supply the *error_direction* information.

   The *error_direction* defaults to a value of CM_RECEIVE_ERROR. To override the default, a program can issue the Set_Error_Direction call prior to issuing Send_Error.

   Once changed, the new *error_direction* value remains in effect until the program changes it again. Therefore, a program should issue Set_Error_Direction before issuing Send_Error for a conversation in **Send-Pending** state.

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

2. If the conversation is in **Send-Pending** state and the program issues a Send_Error call, CPI Communications examines the *error_direction* characteristic and notifies the partner program accordingly:

   - If *error_direction* is set to CM_RECEIVE_ERROR, the partner program receives a *return_code* of CM_PROGRAM_ERROR_PURGING. This indicates that an error at the remote program occurred in the data before the remote program received send control.

   - If *error_direction* is set to CM_SEND_ERROR, the partner program receives a *return_code* of CM_PROGRAM_ERROR_NO_TRUNC. This indicates that an error at the remote program occurred in the send processing after the remote program received send control.

3. If a *return_code* other than CM_OK is returned on the call, the *error_direction* conversation characteristic is unchanged.

Related Information

"Example 7: Error Direction and Send-Pending State" on page 82 provides an example program using Set_Error_Direction.

"Send_Error (CMSERR)" on page 259 provides more information on reporting errors.

"Send-Pending State and the error_direction Characteristic" on page 726 provides more information on the *error_direction* characteristic.

Set_Fill (CMSF)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

Set_Fill (CMSF) is used by a program to set the *fill* characteristic for a given conversation. Set_Fill overrides the value that was assigned when the Initialize_Conversation, Accept_Conversation, or Initialize_For_Incoming call was issued.

Programs use the Set_Fill call to indicate to the local system the condition that will be used to determine when sufficient data is available to complete a Receive call.

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

Format

```
CALL CMSF(conversation_ID,
          fill,
          return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

fill *(input)*
Specifies whether the program is to receive data in terms of the logical-record format of the data. The *fill* variable can have one of the following values:

- CM_FILL_LL
  Specifies that the program is to receive one complete or truncated logical record, or a portion of the logical record that is equal to the length specified by the *requested_length* parameter of the Receive call.
- CM_FILL_BUFFER
  Specifies that the program is to receive data independent of its logical-record format. The amount of data received will be equal to or less than the length specified by the *requested_length* parameter of the Receive call. The amount is less than the requested length when the program receives the end of the data.

return_code *(output)*

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

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *conversation_type* specifies CM_MAPPED_CONVERSATION.
  - The *fill* specifies an undefined value.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. The *fill* value provided (for a basic conversation) is used on all subsequent Receive calls for the specified conversation until changed by the program with another Set_Fill call.
2. If a *return_code* other than CM_OK is returned on the call, the *fill* conversation characteristic is unchanged.

Related Information

"Receive (CMRCV)" on page 213 provides more information on how the *fill* characteristic is used for basic conversations.

Set_Initialization_Data (CMSID)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32

Set_Initialization_Data (CMSID) is used by a program to set the *initialization_data* and *initialization_data_length* conversation characteristics to be sent to the remote program for a given conversation. Set_Initialization_Data overrides the values that were assigned when the Initialize_Conversation, Accept_Conversation, Initialize_For_Incoming, or Accept_Incoming call was issued.

The *initialization_data* conversation characteristic contains program-supplied data that is sent to, or received from, the remote program during conversation initialization. It may be specified prior to Allocate by the conversation initiator. When the conversation is using an OSI TP CRM it may also be specified by the recipient of the conversation immediately following Accept_Conversation or Accept_Incoming.

Format

```
CALL CMSID(conversation_ID,
           initialization_data,
           initialization_data_length,
           return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

initialization_data *(input)*
Specifies the initialization data that is to be passed to the remote program during conversation startup.

initialization_data_length *(input)*
Specifies the length of the initialization data. The length can be from 0 to 10000 bytes. If zero, the *initialization_data* parameter is ignored.

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The conversation is not in **Initialize** state, **Receive** state (for a half-duplex conversation), or **Send-Receive** state (for a full-duplex conversation).

- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned identifier.
  - The *initialization_data_length* specifies a value greater than 10000 or less than zero.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

1. Initialization data is data that the program initiating a conversation (using the Initialize_Conversation call) can choose to send to the remote program. The initiator issues a Set_Initialization_Data call between the Initialize_Conversation and Allocate calls to specify the initialization data to be sent by the Allocate call. Following the successful completion of either an Accept_Conversation or Accept_Incoming call, the remote program issues an Extract_Initialization_Data call to extract the initialization data.
2. When the conversation is allocated using an OSI TP CRM, the remote program may issue a Set_Initialization_Data call to identify data that is to be sent to the initiating program on its next call following its acceptance of the conversation. To avoid overwriting initialization data from the initiating program, the recipient must issue the Extract_Initialization_Data call to extract the incoming initialization data before issuing the Set_Initialization_Data call. The Set_Initialization_Data must be issued before any other call on the conversation, except Set_* or Extract_* calls. To extract the data from the remote program, the initiating program issues the Extract_Initialization_Data call after receiving a *control_information_received* value of CM_ALLOCATE_CONFIRMED_WITH_DATA or CM_ALLOCATE_REJECTED_WITH_DATA.
3. If a *return_code* other than CM_OK is returned on the call, the *initialization_data* and *initialization_data_length* conversation characteristics remain unchanged.

Related Information

"Extract_Initialization_Data (CMEID)" on page 168 describes the Extract_Initialization_Data call.

"Extract_Mapped_Initialization_Data (CMEMID)" on page 170 describes information about the Extract_Mapped_Initialization_Data call.

"Set_Mapped_Initialization_Data (CMSMID)" on page 318 describes information about the Set_Mapped_Initialization_Data call.

Set_Join_Transaction (CMSJT)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32

Programs use the Set_Join_Transaction call to set the *join_transaction* characteristic for a conversation. Set_Join_Transaction overrides the value assigned when the Accept_Conversation or Accept_Incoming call was issued.

Note: The *join_transaction* characteristic is only meaningful in conjunction with the X/Open TX resource recovery interface.

Format

```
CALL CMSJT(conversation_ID,
           join_transaction,
           return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation ID.

join_transaction *(input)*
Specifies if the subordinate implicitly joins the transaction after receiving a join transaction request from the superior. The *join_transaction* characteristic can have one of the following values:

- CM_JOIN_IMPLICIT
  Specifies that the subordinate automatically joins the transaction when receiving a join transaction request from the superior.
- CM_JOIN_EXPLICIT
  Specifies that the subordinate must join the transaction explicitly when receiving a join transaction request from the superior.

return_code *(output)*
Specifies the results of the call execution. The *return_code* can be one of the following:

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  – The *conversation_ID* specifies an unassigned identifier.
  – The *join_transaction* specifies an undefined value.
  – The program is not the subordinate for the conversation.
  – The *transaction_control* characteristic is set to CM_CHAINED_TRANSACTIONS.

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

State Changes

This call does not cause a state change.

Usage Notes

1. If a *return_code* other than CM_OK is returned on the call, the *join_transaction* conversation characteristic remains unchanged.
2. This call can be issued only by the subordinate program of a conversation.
3. This call can be issued in all states except **Reset** and **Initialize**. It should be issued in the **Initialize_Incoming** state, so that it has an effect at the following Accept_Incoming call. In all other states, it is allowed only if the *transaction_control* characteristic has the value CM_UNCHAINED_TRANSACTION.
4. If a program wants to use CM_JOIN_EXPLICIT, it should extract the *transaction_control* characteristic after a successful Accept_Incoming call. If the value is CM_CHAINED_TRANSACTIONS, the program should join the transaction by issuing a tx_begin() call. If the value is CM_UNCHAINED_TRANSACTIONS, the program is informed with a CM_JOIN_TRANSACTION *status_received* value if it is to join the transaction. In any case, the program may first do any local work that is not to be included in the remote program's transaction before joining the transaction.
5. This call does not apply to any previous call operation still in progress.
6. This call is not associated with any conversation queue. When a conversation uses queue-level non-blocking, the call does not return CM_OPERATION_NOT_ACCEPTED on the conversation.

Related Information

“Joining a Transaction” on page 61 discusses how a program can join a transaction.

“TX Extensions for CPI Communications” on page 66 discusses the TX extensions for CPI Communications.

Set_Log_Data (CMSLD)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

Set_Log_Data (CMSLD) is used by a program to set the *log_data* and *log_data_length* characteristics for a given conversation. Set_Log_Data overrides the values that were assigned when the Initialize_Conversation, Accept_Conversation, or Initialize_For_Incoming call was issued.

The *log_data* conversation characteristic contains program-supplied information that will be sent to the remote system when an error occurs. It may be helpful in determining what caused the error condition.

Note: When an LU 6.2 CRM is being used, the *log_data* characteristic is used only on basic conversations.

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

For Networking Services for Windows, this call does update the *log_data* conversation characteristic. *log_data* is written to the trace log upon entry to Set_Log_Data and Deallocate. Networking Services for Windows does not send *log_data* to the remote program.

An AIX application program can issue a Set_Log_Data call; however, prior to Version 3 Release 1, this call performs no operation. AIX does not log or transmit the data until Version 3 Release 1 or later.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

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

log_data_length *(input)*
Specifies the length of the program-unique error information. The length can be from 0 to 512 bytes. If zero, the *log_data_length* characteristic is set to zero

(effectively setting the *log_data* characteristic to the null string), and the *log_data* parameter on this call is ignored.

return_code *(output)*

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

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
  This value can be one of the following:
  – The *conversation_ID* specifies an unassigned conversation identifier.
  – The *conversation_type* is set to CM_MAPPED_CONVERSATION and the conversation is using an LU 6.2 CRM.
  – The *log_data_length* specifies a value less than 0 or greater than 512.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. If the *log_data* characteristic contains data (as a result of a Set_Log_Data call), log data will be sent to the remote system under any of the following conditions:
   - When the local program issues a Send_Error call and the conversation is using an LU 6.2 CRM
   - When the local program issues a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND
   - When the local program issues a Send_Data call with *send_type* set to CM_SEND_AND_DEALLOCATE and *deallocate_type* set to CM_DEALLOCATE_ABEND
2. The system resets the *log_data* and *log_data_length* characteristics to their initial (null) values after sending the log data. Therefore, the *log_data* is sent to the remote system only once even though an error indication may be issued several times. See usage note 1 for conditions when log data is sent.
3. Specify *log_data* using the local system's native encoding. When the log data is displayed on the partner system, it will be displayed in that system's native encoding.
4. If a *return_code* other than CM_OK is returned on the call, the *log_data* and *log_data_length* conversation characteristics are unchanged.

Related Information

“Automatic Conversion of Characteristics” on page 41 provides further information on the automatic conversion of the *log_data* parameter.

“Send_Error (CMSERR)” on page 259 and “Deallocate (CMDEAL)” on page 143 provide further discussion on how the *log_data* characteristic is used.

Set_Mapped_Initialization_Data (CMSMID)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32

A program uses the Set_Mapped_Initialization_Data (CMSMID) call to set the *initialization_data* and *initialization_data_length* conversation characteristics to be sent to the remote program for a given conversation. The *map_name* identifies the data so the underlying map routine can properly transform it for transmission to the remote partner. Set_Mapped_Initialization_Data overrides the values that were assigned when the Initialize_Conversation, Accept_Conversation, Initialize_For_Incoming, or Accept_Incoming call was issued.

Format

```
CALL CMSMID(conversation_ID,
            map_name,
            map_name_length, initialization_data,
            initialization_data_length,
            return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

map_name (input)
Specifies the mapping function used to encode the data record. The length of the identifier is 0 to 64 bytes.

map_name_length (input)
Specifies the length of the map_name. The length can range from 0 to 64 bytes. If the length is zero, the map_name parameter is ignored.

initialization_data *(input)*
Specifies the initialization data that is to be passed to the remote program during conversation start up.

initialization_data_length *(input)*
Specifies the length of the initialization data. If zero, the *initialization_data* parameter is ignored.

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

- CM_OK
- CM_UNKNOWN_MAP_NAME_REQUESTED
  The supplied map name is unknown to the map routine. No data was sent.
- CM_MAP_ROUTINE_ERROR
  The map routine encountered a problem with the user data. No data was sent.

- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_STATE_CHECK

This value indicates one of the following:

- The conversation is not in initialize state, Receive state (for a half-duplex conversation), or **Send-Receive** state (for a full-duplex conversation).
- CM_PROGRAM_PARAMETER_CHECK

Value indicates one of the following:

  - The *conversation_ID* specifies an unassigned identifier.
  - The *initialization_data_length* specifies a value greater than 10000 or less than 0.
  - The *conversation_type* characteristic is set to CM_BASIC_CONVERSATION.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

1. Initialization data is data that the program initiating a conversation, using the Initialize_Conversation call, can choose to send to the remote program. The initiator issues a Set_Mapped_Initialization Data call between the Initialize_Conversation and Allocate calls to specify the initialization to be sent by the Allocate call. The *map_name* parameter is set to identify the data so that the underlying map routine can properly handle it. Following the successful completion of either an Accept_Conversation or Accept_Incoming call, the remote program issues an Extract_Mapped_Initialization_Data call to extract the initialization data.
2. The remote program may issue a Set_Mapped_Initialization_Data call to identify data that is to be sent to the initiating program on its next call following its acceptance of the conversation. To avoid overwriting initialization data from the initiating program, the recipient must issue the Extract_Mapped_Initialization_Data call to extract the incoming initialization data before issuing the Set_Mapped_Initialization_Data call. The Set_Mapped_Initialization Data must be issued before any other call on the conversation, except Set or Extract calls. To extract the data from the remote program, the initiating program issues the Extract_Mapped_Initialization_Data call after receiving a *control_information_received* value of CM_ALLOCATE_CONFIRMED_WITH_DATA or CM_ALLOCATE_REJECTED_WITH_DATA.
3. If a *return_code* other than CM_OK is returned on the call, the *initialization_data* and *initialization_data_length* conversation characteristics remain unchanged.
4. If the map routine is unable to process the user data either because the user identified an incorrect *map_name* or the map routine encountered an error during processing the *return_code* is set appropriately. Nothing is sent nor is the conversation aborted. Control is given back to the program for appropriate action.

5. Because of the work the local map routine performs, the real amount of data sent by the CRM may be larger than 10000 bytes, even if the initialization_data_length is less than 10000. In this case the remote system may not be able to process the data. Therefore, the maximum value of *initialization_data_length* depends on the map routine and the abilities of the remote system.
6. When the conversation is allocated using the LU 6.2 CRM, initialization data is sent as PIP data.

Related Information

"Extract_Initialization_Data (CMEID)" on page 168 describes the Extract_Initialization_Data call.

"Set_Initialization_Data (CMSID)" on page 312 describes the Set_Initialization_Data call.

"Extract_Mapped_Initialization_Data (CMEMID)" on page 170 describes the Extract_Mapped_Initialization_Data call.

Set_Mode_Name (CMSMN)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

Set_Mode_Name (CMSMN) is used by a program to set the *mode_name* and *mode_name_length* characteristics for a conversation. Set_Mode_Name overrides the current values that were originally acquired from the side information using the *sym_dest_name*.

Issuing this call does not change the values in the side information. It only changes the *mode_name* and *mode_name_length* characteristics for this conversation.

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

The *mode_name* conversation characteristic is used during conversation initialization to select a session with the appropriate attributes; such as, cost, bandwidth, and so on.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

mode_name *(input)*
Specifies the mode name designating the network properties for the logical connection to be allocated for the conversation. The network properties include, for example, the class of service to be used, and whether data is to be enciphered.

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

CPI Communications applications in CICS cannot be SNA service programs and, therefore, cannot allocate on the mode names SNASVCMG or CPSVCMG. If they attempt to do this they will get CM_PARAMETER_ERROR.

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

return_code *(output)*

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

- CM_OK
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is not in **Initialize** state.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *mode_name_length* specifies a value less than zero or greater than eight.
  - The *partner_ID* characteristic is set to a non-null value.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. Specification of a mode name that is not recognized by the system is not detected on this call. It is detected on the subsequent Allocate call.
2. Specify *mode_name* using the local system's native encoding. CPI Communications automatically converts the *mode_name* from the native encoding where necessary.
3. If a *return_code* other than CM_OK is returned on the call, the *mode_name* and *mode_name_length* conversation characteristics are unchanged.

Related Information

“Side Information” on page 23 further discusses the *mode_name* conversation characteristic.

“Automatic Conversion of Characteristics” on page 41 provides further information on the automatic conversion of the *mode_name* parameter.

“SNA Service Transaction Programs” on page 727 discusses SNA service transaction programs.

Set_Partner_ID (CMSPID)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32

Programs use the Set_Partner_ID call to specify a *partner_ID* characteristic that will be used to allocate the conversation. Set_Partner_ID overrides the current values for the *partner_ID*, *partner_ID_length*, *partner_ID_type*, and *partner_ID_scope* characteristics that were originally acquired from the side information using the *sym_dest_name*.

Issuing this call does not change the information in the side information. It only changes the *partner_ID*, *partner_ID_length*, *partner_ID_type*, and *partner_ID_scope* characteristics for this conversation.

Format

```
CALL CMSPID(conversation_ID,
            partner_ID_type,
            partner_ID,
            partner_ID_length,
            partner_ID_scope,
            directory_syntax,
            directory_encoding,
            return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation ID.

partner_ID_type *(input)*
Specifies the type of the *partner_ID* that will be used on this conversation.

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

- CM_DISTINGUISHED_NAME
  The *partner_ID* variable contains a complete distinguished name CPI Communications can use to retrieve destination information from a distributed directory.
- CM_LOCAL_DISTINGUISHED_NAME
  The *partner_ID* variable contains a local distinguished name. When combined with a system-specific prefix, the local DN creates a complete DN for access of destination information from a distributed directory.
- CM_PROGRAM_FUNCTION_ID
  The *partner_ID* variable contains a program function identifier that CPI Communications can use to search the distributed directory and obtain the partner binding.

- CM_OSI_TPSU_TITLE_OID
  The *partner_ID* variable contains an object identifier for an OSI TPSU title. On systems supporting attribute_type searches, CPI Communications will search the directory for objects containing attributes with an attribute_type of TPSU_ENTRY and an attribute_value equal to the specified partner_ID.
- CM_PROGRAM_BINDING
  The *partner_ID* variable contains a program binding for the partner program. For more information on the format of a program binding, see "Program Binding" on page 658.

partner_ID *(input)* Specifies the *partner_ID* to be used in allocating the conversation.

partner_ID_length *(input)* Specifies the length of the *partner_ID*. The length can be from 0 to 32767 bytes. If the *partner_ID_length* is zero (effectively setting the *partner_ID* characteristic to the null string), the *partner_ID* parameter on this call is ignored.

partner_ID_scope *(input)* Specifies the scope of the search in the use of the *partner_ID*. The *partner_ID_scope* variable only has meaning when the *partner_ID* is of type CM_DISTINGUISHED_NAME. It can have one of the following values:

- CM_EXPLICIT
  CPI Communications should only use the destination information contained in the program installation object identified by the distinguished name.
- CM_REFERENCE
  CPI Communications should access the program binding contained in the program installation object identified by the distinguished name. If CPI Communications is unable to establish a logical connection with the partner CRM using this binding, CPI Communications should use the PFID contained in the program installation object (if available) to search for and attempt to use other program installation objects.

directory_syntax *(input)*
Specifies the syntax of the distributed directory that CPI Communications will access with the *partner_ID*. This parameter is ignored if the *partner_ID_type* is CM_PROGRAM_BINDING. *directory_syntax* can have one of the following values:

- CM_DEFAULT_SYNTAX
  The DN or PFID specified can be used with a directory conforming to the default directory syntax for the implementation.
- CM_DCE_SYNTAX
  The DN or PFID specified can be used with a directory conforming to DCE directory syntax guidelines.
- CM_XDS_SYNTAX
  The DN or PFID specified can be used with a directory conforming to XDS directory syntax guidelines.
- CM_NDS_SYNTAX
  The DN or PFID specified can be used with a directory conforming to Novell's NDS directory syntax guidelines.

directory_encoding *(input)*
Specifies the encoding rule of the distributed directory that CPI Communications will access with the *partner_ID*. This parameter is ignored if the

partner_ID_type is CM_PROGRAM_BINDING. *directory_encoding* can have one of the following values:

- CM_DEFAULT_ENCODING
  The DN or PFID specified is encoded in 8-bit locally-defined format.
- CM_UNICODE_ENCODING
  The DN or PFID specified is encoded in 16-bit unicode format.

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is not in **Initialize** state.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *partner_ID_type* parameter specifies an undefined value.
  - The *partner_ID_scope* parameter specifies an undefined value.
  - The *partner_ID_type* parameter specifies CM_PROGRAM_BINDING and the *partner_ID_length* parameter specifies a value less than 0 or greater than 32767.
  - The *partner_ID_type* parameter specifies CM_DISTINGUISHED_NAME or CM_PROGRAM_FUNCTION_ID and the *partner_ID_length* parameter specifies a value less than 0 or greater than 1024.
  - The *directory_syntax* parameter specifies an undefined value.
  - The *directory_encoding* parameter specifies an undefined value.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This program does not cause a state change.

Usage Notes

1. If the *partner_ID* variable is not null, it is used by CPI Communications to allocate the conversation. Any other destination information (specified in side information or using Set calls) is ignored.
2. If the *partner_ID* characteristic is null, the *partner_ID* characteristic will not be used to allocate the conversation. Instead, CPI Communications will use any destination information provided by side information or Set calls.
3. If a *return_code* other than CM_OK is returned on the call, the characteristics for the conversation are unchanged.
4. If the program sets *partner_ID_type* to CM_PROGRAM_FUNCTION_ID, the *partner_ID* parameter will be used as a PFID to search the distributed directory. CPI Communications uses the PFID in conjunction with a default DN. The specification of a default DN is system-specific and its use depends on the contents of the directory object pointed to:
   - If the default DN points to one of the directory objects defined in Appendix F, “CPI Communications Extensions for Use with DCE Directory,”

CPI Communications will use the contents of the object to direct its search. See "Scenarios for Use of CNSI" on page 742 for examples.

- If the default DN does not point to a directory object recognized by CPI Communications, the default DN will be used as a starting point for a search of the directory tree for objects containing a PFID with a value equal to the value specified in *partner_ID*.

Related Information

"Distributed Directory" on page 25 explains the terms and concepts required for use of a distributed directory.

"Example 14: Using the Distributed Directory to Find the Partner Program" on page 96 provides a sample scenario of a program using the Set_Partner_ID call.

"Extract_Partner_ID (CMEPID)" on page 177 can be used to extract the *partner_ID* characteristic.

"Program Binding" on page 658 describes the format of a program binding.

Set_Partner_LU_Name (CMSPLN)

LU 6.2

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

Set_Partner_LU_Name (CMSPLN) is used by a program to set the *partner_LU_name* and *partner_LU_name_length* characteristics for a conversation. Set_Partner_LU_Name overrides the current values that were originally acquired from the side information using the *sym_dest_name*.

Issuing this call does not change the information in the side information. It only changes the *partner_LU_name* and *partner_LU_name_length* characteristics for this conversation.

The *partner_lu_name* conversation characteristic contains the SNA LU name that identifies the location of the remote program. It may be set by the conversation initiator to identify the location of the remote program.

Notes:

1. A program cannot issue Set_Partner_LU_Name after an Allocate call is issued. Only the program that initiated the conversation (issued the Initialize_Conversation call) can issue Set_Partner_LU_Name.
2. The *partner_LU_name* characteristic is used only by an LU 6.2 CRM.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

partner_LU_name *(input)*
Specifies the name of the remote LU at which the remote program is located. This LU name is any name by which the local system knows the remote LU for purposes of allocating a conversation.

On MVS, a blank partner LU name can be specified. If Allocate (CMALLC) is called while the partner LU name is set to blanks, the system considers the partner LU name to be the name of the local LU from which CMALLC is called. MVS also allows a blank partner LU name to be specified as an entry in the side information.

partner_LU_name_length *(input)*

Specifies the length of the partner LU name. The length can be from 1 to 17 bytes.

On MVS, the length of a blank partner LU name can also be 1 to 17 bytes.

return_code *(output)*

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *partner_LU_name_length* is set to a value less than 1 or greater than 17.
  - The *partner_ID* characteristic is set to a non-null value.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is not in **Initialize** state.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. Specify *partner_LU_name* using the local system's native encoding. CPI Communications automatically converts the *partner_LU_name* from the native encoding where necessary.
2. If a *return_code* other than CM_OK is returned on the call, the *partner_LU_name* and *partner_LU_name_length* conversation characteristics are unchanged.

Related Information

"Side Information" on page 23 and notes 4 and 5 of Table 61 on page 650 provide further discussion of the *partner_LU_name* conversation characteristic.

"Automatic Conversion of Characteristics" on page 41 provides further information on the automatic conversion of the *partner_LU_name* parameter.

Set_Prepare_Data_Permitted (CMSPDP)

OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32

Set_Prepare_Data_Permitted (CMSPDP) is used by a program to set the *prepare_data_permitted* characteristic for a given conversation. Set_Prepare_Data_Permitted overrides the value that was assigned when the Initialize_Conversation call was issued. The subordinate program on the conversation cannot issue the Set_Prepare_Data_Permitted call.

The superior program uses the Set_Prepare_Data Permitted call to indicate whether the subordinate may send data following receipt of a *take_commit* notification.

Note: The *prepare_data_permitted* characteristic is used only by an OSI TP CRM and only for a half-duplex conversation.

Format

```
CALL CMSPDP(conversation_ID,
            prepare_data_permitted,
            return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

prepare_data_permitted *(input)*
Specifies whether the superior program wants to allow the subordinate to send data following the receipt of a take-commit notification. The *prepare_data_permitted* variable can have one of the following values:

- CM_PREPARE_DATA_NOT_PERMITTED
  Specifies the subordinate will not be permitted to send data following the receipt of a take-commit notification.
- CM_PREPARE_DATA_PERMITTED
  Specifies the subordinate will be permitted to send data following the receipt of a take-commit notification.

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned identifier.

- The *prepare_data_permitted* specifies CM_PREPARE_DATA_PERMITTED and the conversation is using an LU 6.2 CRM.
- The *prepare_data_permitted* specifies an undefined value.
- The *sync_level* is set to CM_NONE or CM_CONFIRM.
- The program is not the superior for the conversation.

- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is in **Initialize-Incoming** state.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

1. If a *return_code* other than CM_OK is returned on the call, the *prepare_data_permitted* conversation characteristic remains unchanged.
2. When the Prepare call is issued with the *prepare_data_permitted* characteristic set to CM_PREPARE_DATA_PERMITTED, the subordinate program is notified that it is permitted to send data through a take-commit notification that ends in a *status_received* value of CM_TAKE_COMMIT_DATA_OK, CM_TAKE_COMMIT_SEND_DATA_OK, or CM_TAKE_COMMIT_DEALLOC_DATA_OK.

Related Information

See “Prepare (CMPREP)” on page 205 for a description of the Prepare call.

Set_Prepare_To_Receive_Type (CMSPTR)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

Set_Prepare_To_Receive_Type (CMSPTR) is used by a program to set the *prepare_to_receive_type* characteristic for a conversation. This call overrides the value that was assigned when the Initialize_Conversation, Accept_Conversation, or Initialize_For_Incoming call was issued.

Programs use the Set_Prepare_To_Receive_Type call to identify any additional CPI-C functions (such as, Flush and Confirm) that are to be done in conjunction with the Prepare_To_Receive call.

Note: The *prepare_to_receive_type* characteristic is used only for a half-duplex conversation.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

prepare_to_receive_type *(input)*
Specifies the type of prepare-to-receive processing to be performed for this conversation. The *prepare_to_receive_type* variable can have one of the following values:

- CM_PREP_TO_RECEIVE_SYNC_LEVEL
  Perform the prepare-to-receive based on one of the following *sync_level* settings:
  - If *sync_level* is set to CM_NONE, or if *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM but the conversation is not currently included in a transaction, execute the function of the Flush call and enter **Receive** state.
  - If *sync_level* is set to CM_CONFIRM, or if *sync_level* is set to CM_SYNC_POINT but the conversation is not currently included in a transaction, execute the function of the Confirm call and if successful (as indicated by a return code of CM_OK on the Prepare_To_Receive call, or a return code of CM_OK on the Send_Data call with *send_type* set to CM_SEND_AND_PREP_TO_RECEIVE), enter **Receive** state. If Confirm is not successful, the state of the conversation is determined by the return code.
  - If *sync_level* is set to CM_SYNC_POINT and the conversation is included in a transaction, enter **Defer-Receive** state until the program issues a

resource recovery commit or backout call, or until the program issues a Confirm or Flush call for this conversation. If the commit or Confirm call is successful or if a Flush call is issued, the conversation then enters **Receive** state. If the backout call is successful, the conversation returns to its state at the previous sync point. Otherwise, the state of the conversation is determined by the return code.
  - If *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, and the conversation is included in a transaction, enter **Defer-Receive** state until the program issues a resource recovery commit or backout call, or until the program issues a Flush call for this conversation. If the commit call is successful or if a Flush call is issued, the conversation then enters **Receive** state. If the backout call is successful, the conversation returns to its state at the previous sync point. Otherwise, the state of the conversation is determined by the return code.
- CM_PREP_TO_RECEIVE_FLUSH
  Execute the function of the Flush call and enter **Receive** state.
- CM_PREP_TO_RECEIVE_CONFIRM
  Execute the function of the Confirm call and if successful (as indicated by a return code of CM_OK on the Prepare_To_Receive call, or a return code of CM_OK on the Send_Data call with *send_type* set to CM_SEND_AND_PREP_TO_RECEIVE), enter **Receive** state. If it is not successful, the state of the conversation is determined by the return code.

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

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *prepare_to_receive_type* is set to an undefined value.
  - The *prepare_to_receive_type* is CM_PREP_TO_RECEIVE_CONFIRM, but the conversation is assigned with *sync_level* set to CM_NONE or CM_SYNC_POINT_NO_CONFIRM.
  - The *send_receive_mode* of the conversation is CM_FULL_DUPLEX.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is in **Initialize-Incoming** state.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

If a *return_code* other than CM_OK is returned on the call, the *prepare_to_receive_type* conversation characteristic is unchanged.

Related Information

"Example 4: The Receiving Program Changes the Data Flow Direction" on page 75 shows an example program using the Prepare_To_Receive call.

"Prepare_To_Receive (CMPTR)" on page 208 discusses how the *prepare_to_receive_type* is used.

Set_Processing_Mode (CMSPM)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
					X*			X

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

Note: The *processing_mode* characteristic is used only for a half-duplex conversation.

X* In Communications Manager/2, this call is supported in Communications Server

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

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

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

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  – The *conversation_ID* specifies an unassigned conversation identifier.
  – The *processing_mode* specifies an undefined value.
  – The *send_receive_mode* of the conversation is CM_FULL_DUPLEX.
  – The program has chosen queue-level non-blocking for the conversation.
- CM_OPERATION_NOT_ACCEPTED
  This value indicates that a previous call operation on the conversation is still in progress.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

1. A program can choose to use conversation-level non-blocking by issuing the Set_Processing_Mode call to set the *processing_mode* characteristic to CM_NON_BLOCKING for a conversation. The processing mode applies to all the subsequent calls on that conversation until it is set otherwise or the conversation ends.

2. If a *return_code* other than CM_OK is returned on the call, the *processing_mode* conversation characteristic is unchanged.

3. When CM_OPERATION_INCOMPLETE is returned from any of the calls listed in Table 7 on page 48, the call operation has not completed. The operation proceeds without blocking the program. The data and buffer areas used in the call are in an indeterminate state and should not be referenced until the operation is completed. For conversations using conversation-level non-blocking, the Wait_For_Conversation call is used to determine when an operation is completed. Each call to Wait_For_Conversation will return the conversation identifier and return code (the *conversation_return_code* value) of a completed operation. It is the responsibility of the program to keep track of the operation being performed by each conversation in order to be able to properly interpret the *conversation_return_code* value.

   If programs place the Wait_For_Conversation call in a procedure other than the accompanying CPI-C call that returned CM_OPERATION_INCOMPLETE, all parameters of that CPI-C call must be in global storage and not in automatic storage. This is important even for parameters the program does not use. For example, if the partner program will never use any call that results in a control_information_received value, other than CM_NO_CONTROL_INFO_RECEIVED, the local program still has to place the *control_information_received* parameter of the Receive call in global storage.

4. Not all language processors support the use of non-blocking operations. See “Programming Language Considerations” on page 111 for language processor restrictions on the use of non-blocking operations.

Related Information

"Concurrent Operations" on page 44 discusses the use of concurrent operations and conversation queues.

"Non-Blocking Operations" on page 47 discusses the use of non-blocking operations.

"Example 13: Accepting Multiple Conversations Using Conversation-Level Non-Blocking Calls" on page 94 shows an example of a program that uses conversation-level non-blocking calls to accept multiple incoming half-duplex conversations.

"Set_Queue_Callback_Function (CMSQCF)" on page 337 describes how to set a callback function and related information for a conversation queue.

"Set_Queue_Processing_Mode (CMSQPM)" on page 340 describes how to set the processing mode for a conversation queue.

"Wait_For_Conversation (CMWAIT)" on page 369 describes the use of Wait_For_Conversation to wait for completion of a conversation-level outstanding operation.

Set_Queue_Callback_Function (CMSQCF)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
				X				X

Set_Queue_Callback_Function (CMSQCF) is used to set a callback function and a user field for a given conversation queue and to set the queue's processing mode to CM_NON_BLOCKING.

Format

```
CALL CMSQCF(conversation_ID,
            conversation_queue,
            callback_function,
            user_field,
            return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

conversation_queue *(input)*
Specifies the conversation queue on which completion of a call operation will invoke the callback function. The *conversation_queue* can have one of the following values:

- CM_INITIALIZATION_QUEUE
- CM_SEND_QUEUE
- CM_RECEIVE_QUEUE
- CM_SEND_RECEIVE_QUEUE
- CM_EXPEDITED_SEND_QUEUE
- CM_EXPEDITED_RECEIVE_QUEUE

callback_function *(input)*
Specifies a callback function to be set for the identified queue.

user_field *(input)*
Specifies a user field to be associated with the identified queue.

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* contains an unassigned conversation identifier.
  - The *conversation_queue* specifies a value that is not defined for the *send_receive_mode* conversation characteristic.

  - The program has chosen conversation-level non-blocking for the conversation.
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The *conversation_queue* parameter is set to CM_INITIALIZATION_QUEUE, and the conversation is not in **Initialize** or **Initialize-Incoming** state.
  - The *conversation_queue* parameter is set to CM_SEND_QUEUE, CM_RECEIVE_QUEUE, CM_SEND_RECEIVE_QUEUE, CM_EXPEDITED_SEND_QUEUE, or CM_EXPEDITED_RECEIVE_QUEUE, and the conversation is in **Initialize-Incoming** state.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. Because of requiring support of passing a callback function as a parameter, the call is supported by the C programming language only.
2. A program can choose to use queue-level non-blocking by issuing the Set_Queue_Callback_Function call (or the Set_Queue_Processing_Mode call) for a conversation queue. When the call completes successfully, the *processing_mode* characteristic becomes meaningless to the conversation.
3. The call is associated with the queue specified in the *conversation_queue* parameter.
4. The program can issue the call for a conversation queue that is defined for the current send-receive mode. The defined queues for each send-receive mode are listed in Table 17 on page 342 under the "Usage Notes" for the Set_Queue_Processing_Mode call.

   In the special case when the conversation is in **Initialize-Incoming** state, the *send_receive_mode* characteristic has no defined value. The program can issue the call only for the Initialization queue.
5. Until the program sets the processing mode for a conversation queue (or chooses conversation-level non-blocking), all the calls associated with that queue are processed in blocking mode.
6. The call sets the processing mode of the identified queue to CM_NON_BLOCKING. The processing mode applies to all subsequent calls to the queue until the processing mode is set to CM_BLOCKING using a Set_Queue_Processing_Mode (CMSQPM) call or the conversation ends.
7. Once set for the identified queue, the callback function and user field will be associated with all subsequent outstanding operations on the queue until they are set differently, a Set_Queue_Processing_Mode (CMSQPM) call is issued, or the conversation ends.
8. When CM_OPERATION_INCOMPLETE is returned from any of the calls listed in Table 7 on page 48, the call operation has not completed; in this case, CPI Communications will return the result of the completed operation in the return_code input parameter to the callback function ***rather than overlaying*** the CM_OPERATION_INCOMPLETE value returned for the incomplete call (but

see the Note below for an exception for Windows 3.x). The operation proceeds without blocking the program. The data and buffer areas used in the call are in an indeterminate state and should not be referenced until the operation is completed.

9. A callback function is a user-defined routine and has three input parameters, *user_field* (as passed in the CMSQCF call), *call_ID* (identifying the completed call), and *return_code* (in which CPI Communications places the return code specifying the final result of the completed operation). The callback function is used to handle completion of an outstanding operation.

   If a callback function is set for a conversation queue, the function is invoked when an outstanding operation on the queue completes. The user field, as specified by the program, and call ID and final return code for the completed operation can then be passed to the callback function.

   Note: An exception is the Microsoft Windows** environment. The *callback_function* parameter is ignored in Windows, and the *user_field* parameter contains a pointer to a structure of type MSG, as defined by Windows. The structure contains at least a window handle, a message number, a word value (in which CPI Communications places the return code specifying the result of the completed operation — except for Windows 3.x, where ***instead*** it overlays the CM_OPERATION_INCOMPLETE value in the *return_code* referenced in the completed call), and a doubleword value. When an outstanding operation completes, CPI Communications will use these fields as parameters to call the Windows PostMessage function. Upon catching the message, the program can then invoke a routine (taking the word value and doubleword value as input) to handle the completion of the outstanding operation.

10. The last call to set queue-level non-blocking takes precedence. This means that if a CMSQPM is followed by a CMSQCF for the same queue, then the completion of outstanding operations will take place through callback. If a CMSQCF is followed by a CMSQPM for the same queue, then completion of outstanding operations will take place through CMWCMP.

11. The user callback function is defined as follows:

    ```
    void callback_fn( void * user_field, CM_CALL_ID * call_id )
    ```

    Note that the input parameters to the callback function are both pointers rather than values.

Related Information

“Concurrent Operations” on page 44 discusses the use of concurrent operations and conversation queues.

“Non-Blocking Operations” on page 47 discusses the use of non-blocking operations.

“Set_Queue_Processing_Mode (CMSQPM)” on page 340 describes how to set the processing mode for a conversation queue.

“Wait_For_Completion (CMWCMP)” on page 366 describes the use of Wait_For_Completion to wait for completion of an outstanding operation on a conversation queue.

Set_Queue_Processing_Mode (CMSQPM)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
				X	X*			X

Set_Queue_Processing_Mode (CMSQPM) is used to set the processing mode for a given conversation queue. When the *queue_processing_mode* is set to CM_NON_BLOCKING, this call also associates an outstanding-operation identifier (OOID) and a user field with the queue.

X* In Communications Manager/2, this call is supported in Communications Server.

Format

```
CALL CMSQPM(conversation_ID,
            conversation_queue,
            queue_processing_mode,
            user_field,
            OOID,
            return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

conversation_queue *(input)*
Specifies the conversation queue for which the processing mode is to be set by this call. *conversation_queue* can have one of the following values:

- CM_INITIALIZATION_QUEUE
- CM_SEND_QUEUE
- CM_RECEIVE_QUEUE
- CM_SEND_RECEIVE_QUEUE
- CM_EXPEDITED_SEND_QUEUE
- CM_EXPEDITED_RECEIVE_QUEUE

queue_processing_mode *(input)*
Specifies the processing mode to be used for the identified queue. *queue_processing_mode* can have one of the following values:

- CM_BLOCKING
- CM_NON_BLOCKING

user_field *(input)*
Specifies a user field to be associated with the identified queue, when *queue_processing_mode* is set to CM_NON_BLOCKING.

OOID *(output)*
Specifies an outstanding operation identifier assigned to the identified queue, when *queue_processing_mode* is set to CM_NON_BLOCKING.

return_code *(output)*

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* contains an unassigned conversation identifier.
  - The *conversation_queue* specifies a value that is not defined for the *send_receive_mode* conversation characteristic.
  - The *queue_processing_mode* specifies an undefined value.
  - The program has chosen conversation-level non-blocking for the conversation.
- CM_PROGRAM_STATE_CHECK
  This value indicates one of the following:
  - The *conversation_queue* parameter is set to CM_INITIALIZATION_QUEUE, and the conversation is not in **Initialize** or **Initialize-Incoming** state.
  - The *conversation_queue* parameter is set to CM_SEND_QUEUE, CM_RECEIVE_QUEUE, CM_SEND_RECEIVE_QUEUE, CM_EXPEDITED_SEND_QUEUE, or CM_EXPEDITED_RECEIVE_QUEUE, and the conversation is in **Initialize-Incoming** state.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. A program can choose to use queue-level non-blocking by issuing the Set_Queue_Processing_Mode call (or the Set_Queue_Callback_Function call) for a conversation queue. When the call completes successfully, the *processing_mode* characteristic becomes meaningless to the conversation.

2. The call is associated with the queue specified in the *conversation_queue* parameter.

3. The program can issue the call for a conversation queue that is defined for the current send-receive mode. The defined queues for each send-receive mode are listed in Table 17 on page 342.

   In the special case when the conversation is in **Initialize-Incoming** state, the *send_receive_mode* characteristic has no defined value. The program can issue the call only for the Initialization queue.

Table 17. Full-Duplex and Half-Duplex Conversation Queues

Send-Receive Mode	Conversation Queues
Full-Duplex	Initialization Send Receive Expedited-Send Expedited-Receive
Half-Duplex	Initialization Send-Receive Expedited-Send Expedited-Receive

4. Until the program sets the processing mode for a conversation queue (or chooses conversation-level non-blocking), all the calls associated with that queue are processed in blocking mode.

5. If a return code other than CM_OK is returned on the call, the processing mode of the specified queue is unchanged.

6. Once a processing mode is set for a conversation queue, the processing mode applies to all the subsequent calls associated with the queue until it is set differently using the call or until the conversation ends.

7. If the queue processing mode is CM_BLOCKING, no OOID is returned on the call. Therefore, the program should ignore the *OOID* parameter.

8. If the queue processing mode is CM_NON_BLOCKING, an OOID is returned on the call. The OOID will be associated with all subsequent outstanding operations on the queue until the processing mode is set to CM_BLOCKING, a Set_Queue_Callback_Function (CMSQCF) call is issued, or the conversation ends.

9. When the program issues the call for a conversation queue and sets the queue processing mode to CM_NON_BLOCKING for the first time, an OOID is created and set for the queue. The OOID remains with the queue until the conversation ends. Even if the queue processing mode changes several times or a Set_Queue_Callback_Function (CMSQCF) call is issued during the conversation, each Set_Queue_Processing_Mode call to return to CM_NON_BLOCKING reactivates the same OOID for that queue. If a CMSQCF call is issued for a queue that has an OOID associated with it, then the OOID will not be available for wait operations until a CMSQPM call is re-issued for that queue.

10. When CM_OPERATION_INCOMPLETE is returned from any of the calls listed in Table 7 on page 48, the call operation has not completed. The operation proceeds without blocking the program. The data and buffer areas used in the call are in an indeterminate state and should not be referenced until the operation is completed. For conversations using the Set_Queue_Processing_Mode call, the Wait_For_Completion call is used to determine when an outstanding operation is completed.

11. The program may specify a user field on the call when the queue processing mode is CM_NON_BLOCKING. If the program chooses to do so, the user field will be associated with the identified queue along with an OOID. The user field specified by the program will be returned on the *user_field_list* parameter of the Wait_For_Completion (CMWCMP) call when an outstanding operation with the OOID has completed.

12. The last call to set queue-level non-blocking takes precedence. This means that if a CMSQPM is followed by a CMSQCF for the same queue, then the completion of outstanding operations will take place through callback. If a CMSQCF is followed by a CMSQPM for the same queue, then completion of outstanding operations will take place through CMWCMP.

Related Information

"Concurrent Operations" on page 44 discusses the use of concurrent operations and conversation queues.

"Non-Blocking Operations" on page 47 discusses the use of non-blocking operations.

"Example 11: Queue-Level Non-Blocking" on page 90 shows an example of a program that uses queue-level non-blocking.

"Set_Queue_Callback_Function (CMSQCF)" on page 337 describes how to set a callback function and related information for a conversation queue.

"Wait_For_Completion (CMWCMP)" on page 366 describes the use of Wait_For_Completion to wait for completion of an outstanding operation on a conversation queue.

Set_Receive_Type (CMSRT)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

Set_Receive_Type (CMSRT) is used by a program to set the *receive_type* characteristic for a conversation. Set_Receive_Type overrides the value that was assigned when the Initialize_Conversation, Accept_Conversation, or Initialize_For_Incoming call was issued.

Programs use the Set_Receive_Type call to indicate whether the Receive call will return immediately when no data is available, or wait for data to be received.

Format

```
CALL CMSRT(conversation_ID,
           receive_type,
           return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

receive_type *(input)*
Specifies the type of receive to be performed. The *receive_type* variable can have one of the following values:

- CM_RECEIVE_AND_WAIT
  The Receive call is to wait for information to arrive on the specified conversation. If information is already available, the program receives it without waiting.
- CM_RECEIVE_IMMEDIATE
  The Receive call is to receive any information that is available from the specified conversation, but is not to wait for information to arrive.

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

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *receive_type* specifies an undefined value.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

If a *return_code* other than CM_OK is returned on the call, the *receive_type* conversation characteristic is unchanged.

Related Information

"Example 3: The Sending Program Changes the Data Flow Direction" on page 74 discusses how a program can use Set_Receive_Type with a value of CM_RECEIVE_IMMEDIATE.

"Receive (CMRCV)" on page 213 discusses how the *receive_type* characteristic is used.

"Receive_Mapped_Data (CMRCVM)" on page 231 discusses how a program is used to receive mapped partner data.

Set_Return_Control (CMSRC)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

Set_Return_Control (CMSRC) is used to set the *return_control* characteristic for a given conversation. Set_Return_Control overrides the value that was assigned when the Initialize_Conversation call was issued.

Note: A program cannot issue the Set_Return_Control call after an Allocate has been issued for a conversation. Only the program that initiates the conversation (with the Initialize_Conversation call) can issue this call.

The conversation initiator uses the Set_Return_Control call to indicate whether the Allocate call will return before the conversation is assigned to a session, or wait until a session is available.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

return_control *(input)*
Specifies when a program receives control back after issuing a call to Allocate. The *return_control* can have one of the following values:

- CM_WHEN_SESSION_ALLOCATED
  Allocate a logical connection for the conversation before returning control to the program.
- CM_IMMEDIATE

  Allocate a logical connection for the conversation if a logical connection is immediately available and return control to the program with one of the following return codes indicating whether or not a logical connection is allocated.
  - A return code of CM_OK indicates a logical connection was immediately available and has been allocated for the conversation. A logical connection is immediately available when it is active; the logical connection is not allocated to another conversation; and, for an LU 6.2 CRM, the local system is the contention winner for the logical connection.
  - A return code of CM_UNSUCCESSFUL indicates a logical connection is not immediately available. Allocation is not performed.
- CM_WHEN_CONWINNER_ALLOCATED

This value is supported on OS/2 only.
Allocate a contention winner session for the conversation before returning control to the program.

- CM_WHEN_SESSION_FREE
This value is supported on OS/2 only.
Allocate a session for the conversation if a session is available. If no session is available, the local program is willing to wait for session activation. If a session cannot be activated and there are no outstanding session activation requests to satisfy this Allocate, control returns to the program with a return code of CM_ALLOCATE_FAILURE_RETRY.

Table 18. Return Control Options

Return_control value	Con-winner acceptable	Con-loser acceptable	Wait expected
CM_WHEN_SESSION_ALLOCATED	Yes	Yes	Yes, indefinitely
CM_WHEN_CONWINNER_ALLOCATED	Yes	No	Yes, indefinitely
CM_WHEN_SESSION_FREE	Yes	Yes	Yes, while there are outstanding session activation requests
CM_IMMEDIATE	Yes	No	No

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

- CM_OK
- CM_PROGRAM_STATE_CHECK
This value indicates that the conversation is not in **Initialize** state.
- CM_PROGRAM_PARAMETER_CHECK
This value indicates one of the following:
  – The *conversation_ID* specifies an unassigned conversation identifier.
  – The *return_control* specifies an undefined value.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. An allocation error resulting from the local system's failure to obtain a logical connection for the conversation is reported on the Allocate call. An allocation error resulting from the remote system's rejection of the conversation startup request is reported on a subsequent conversation call.

2. For an LU 6.2 CRM, two systems connected by a logical connection may both attempt to allocate a conversation on the logical connection at the same time. This is called contention. Contention is resolved by making one system the contention winner of the session and the other system the contention loser of the session. The contention-winner system allocates a conversation on a session without asking permission from the contention-loser system. Conversely, the contention-loser system requests permission from the

contention-winner system to allocate a conversation on the session, and the contention-winner system either grants or rejects the request. For more information, see *SNA Transaction Programmer's Reference Manual for LU Type 6.2*.

Contention may result in a CM_UNSUCCESSFUL return code for programs specifying CM_IMMEDIATE.

3. The program can modify the function of the Allocate call by specifying a *return_control* characteristic of CM_IMMEDIATE. The result is that control is returned to the program immediately if no session is available. Use of the distributed directory for destination information makes it impossible to guarantee a similar level of function because the CPI Communications implementation's call to the distributed directory might itself make use of a resource that is not immediately available.

   Programs requiring immediate return of control on the Allocate call should consider using non-blocking function, or accessing the directory directly and providing the information as a program binding to CPI Communications using the Set_Partner_ID call.

4. If a *return_code* other than CM_OK is returned on the call, the *return_control* conversation characteristic is unchanged.

Related Information

"Allocate (CMALLC)" on page 124 provides more discussion on the use of the *return_control* characteristic in allocating a conversation.

Set_Send_Receive_Mode (CMSSRM)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
				X	X*			

The Set_Send_Receive_Mode (CMSSRM) call is used by a program to set the *send_receive_mode* characteristic for a conversation. Set_Send_Receive_Mode overrides the value that was assigned when the Initialize_Conversation call was issued.

The *send_receive_mode* conversation characteristic indicates whether the conversation is using half-duplex or full-duplex mode for data transmission. It may be set by the conversation initiator.

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

X* In Communications Manager/2, this call is supported in Communications Server.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

send_receive_mode *(input)*
Specifies the send-receive mode of the conversation.

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

- CM_HALF_DUPLEX
  Specifies the allocation of a half-duplex conversation.
- CM_FULL_DUPLEX
  Specifies the allocation of a full-duplex conversation.

Networking Services for Windows provides a simulated full-duplex conversation support.

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_STATE_CHECK
  This value indicates that the program is not in **Initialize** state.

- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *send_receive_mode* specifies an undefined value.
  - The *sync_level* is set to CM_CONFIRM or CM_SYNC_POINT, and *send_receive_mode* is set to CM_FULL_DUPLEX.
  - The *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, the conversation is using a LU 6.2 CRM, and the *send_receive_mode* specifies CM_HALF_DUPLEX.
  - The *send_type* is set to CM_SEND_AND_CONFIRM or CM_SEND_AND_PREP_TO_RECEIVE, and *send_receive_mode* is set to CM_FULL_DUPLEX.
  - The *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, the conversation is using a LU 6.2 CRM, and the *send_receive_mode* specifies CM_FULL_DUPLEX.
  - The program has selected conversation-level non-blocking by issuing Set_Processing_Mode successfully, and *send_receive_mode* is set to CM_FULL_DUPLEX.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. If a *return_code* other than CM_OK is returned on the call, the *send_receive_mode* characteristic is unchanged.
2. Set_Send_Receive_Mode overrides the value assigned with the Initialize_Conversation call and can only be issued when the program is in **Initialize** state.
3. Networking Services for Windows simulates full-duplex by transparently using two half-duplex conversations. This full-duplex simulation does not allow interoperability with implementations that support true full-duplex conversations.

Related Information

"Send-Receive Modes" on page 19 provides more information on the differences between half-duplex and full-duplex conversations.

"Example 8: Establishing a Full-Duplex Conversation" on page 84 shows an example of how a full-duplex conversation is set up.

"Extract_Send_Receive_Mode (CMESRM)" on page 187 tells how to determine the send-receive mode used for a conversation.

Set_Send_Type (CMSST)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

Set_Send_Type (CMSST) is used by a program to set the *send_type* characteristic for a conversation. Set_Send_Type overrides the value that was assigned when the Initialize_Conversation, Accept_Conversation, or Initialize_For_Incoming call was issued.

Programs use the Set_Send_Type call to identify any additional CPI-C functions (such as, Flush and Confirm) to be done following the transferring of the data to the buffer.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

send_type *(input)*
Specifies what, if any, information is to be sent to the remote program in addition to the data supplied on the Send_Data call, and whether the data is to be sent immediately or buffered.

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

- CM_BUFFER_DATA
  No additional information is to be sent to the remote program. Further, the supplied data might not be sent immediately but, instead, might be buffered until a sufficient quantity is accumulated.
- CM_SEND_AND_FLUSH
  No additional information is to be sent to the remote program. However, the supplied data is sent immediately rather than buffered. Send_Data with *send_type* set to CM_SEND_AND_FLUSH is functionally equivalent to a Send_Data with *send_type* set to CM_BUFFER_DATA followed by a Flush call.
- CM_SEND_AND_CONFIRM (half-duplex conversations only)
  The supplied data is to be sent to the remote program immediately, along with a request for confirmation. Send_Data with *send_type* set to CM_SEND_AND_CONFIRM is functionally equivalent to Send_Data with *send_type* set to CM_BUFFER_DATA followed by a Confirm call.
- CM_SEND_AND_PREP_TO_RECEIVE (half-duplex conversations only)
  The supplied data is to be sent to the remote program immediately, along with send control of the conversation. Send_Data with *send_type* set to

CM_SEND_AND_PREP_TO_RECEIVE is functionally equivalent to Send_Data with *send_type* set to CM_BUFFER_DATA followed by a Prepare_To_Receive call. The action depends on the value of the *prepare_to_receive_type* characteristic for the conversation.

- CM_SEND_AND_DEALLOCATE
  The supplied data is to be sent to the remote program immediately, along with a deallocation notification. Send_Data with *send_type* set to CM_SEND_AND_DEALLOCATE is functionally equivalent to Send_Data with *send_type* set to CM_BUFFER_DATA followed by a call to Deallocate. The action depends on the value of the *deallocate_type* characteristic for the conversation.

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

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *send_type* is set to CM_SEND_AND_CONFIRM and the conversation is assigned with *sync_level* set to CM_NONE or CM_SYNC_POINT_NO_CONFIRM.
  - The *send_type* specifies an undefined value.
  - The *send_type* is set to CM_SEND_AND_CONFIRM or CM_SEND_AND_PREP_TO_RECEIVE and the *send_receive_mode* is set to CM_FULL_DUPLEX.
- CM_PROGRAM_STATE_CHECK
  This value indicates the conversation is in **Initialize-Incoming** state.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

If a *return_code* other than CM_OK is returned on the call, the *send_type* conversation characteristic is unchanged.

Related Information

"Example 4: The Receiving Program Changes the Data Flow Direction" on page 75 shows an example program flow using the Set_Send_Type call.

"Send_Data (CMSEND)" on page 249 discusses how the *send_type* characteristic is used by Send_Data.

"Send_Mapped_Data (CMSNDM)" on page 271 discusses how a program sends mapped data to its partner.

The same function of a call to Send_Data with different values of the *send_type* conversation characteristic in effect can be achieved by combining Send_Data with other calls:

- "Confirm (CMCFM)" on page 133

- “Deallocate (CMDEAL)” on page 143
- “Flush (CMFLUS)” on page 195
- “Prepare_To_Receive (CMPTR)” on page 208

Set_Sync_Level (CMSSL)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

Set_Sync_Level (CMSSL) is used by a program to set the *sync_level* characteristic for a given conversation. The *sync_level* characteristic is used to specify the level of synchronization processing between the two programs. It determines whether the programs support no synchronization, confirmation-level synchronization (using the Confirm and Confirmed CPI Communications calls), or sync-point-level synchronization (using the calls of a resource recovery interface). Set_Sync_Level overrides the value that was assigned when the Initialize_Conversation call was issued.

The *sync_level* conversation characteristic indicates the level of synchronization services provided on the conversation. It may be set by the conversation initiator.

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

Format

CALL CMSSL(*conversation_ID*,
sync_level,
return_code)

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

sync_level *(input)*
Specifies the synchronization level that the local and remote programs can use on this conversation. The *sync_level* can have one of the following values:

- CM_NONE
The programs will not perform confirmation or sync point processing on this conversation. The programs will not issue any calls or recognize any returned parameters relating to synchronization.
- CM_CONFIRM (half-duplex conversations only)
The programs can perform confirmation processing on this conversation. The programs can issue calls and recognize returned parameters relating to confirmation.
- CM_SYNC_POINT (half-duplex conversations only)

  The CM_SYNC_POINT value is only valid on CICS, OS/400, and VM.

  The programs can perform sync point processing on this conversation. The programs can issue resource recovery interface calls and will recognize

returned parameters relating to resource recovery processing. The programs can also perform confirmation processing.

- CM_SYNC_POINT_NO_CONFIRM

  The CM_SYNC_POINT_NO_CONFIRM value is not supported by any of the products.

  The programs can perform sync point processing on this conversation. The programs can issue resource recovery interface calls and will recognize returned parameters relating to resource recovery processing. The programs cannot perform confirmation processing.

Note: If the conversation is using an OSI TP CRM, confirmation of the deallocation of the conversation can be performed with any *sync_level* value.

return_code *(output)*

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

- CM_OK
- CM_PARM_VALUE_NOT_SUPPORTED
  This value indicates that the *sync_level* specifies CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and the value is not supported by the local system.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is not in **Initialize** state.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *sync_level* specifies CM_NONE, the *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, and the conversation is using an LU 6.2 CRM.
  - The *sync_level* specifies CM_NONE, the *send_receive_mode* is set to CM_HALF_DUPLEX, and the *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_CONFIRM.
  - The *sync_level* specifies CM_NONE or CM_SYNC_POINT_NO_CONFIRM, the *send_receive_mode* is set to CM_HALF_DUPLEX, and the *send_type* is set to CM_SEND_AND_CONFIRM.
  - The *sync_level* specifies CM_CONFIRM or CM_SYNC_POINT and the *send_receive_mode* is set to CM_FULL_DUPLEX.
  - The *sync_level* specifies CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the *deallocate_type* is set to CM_DEALLOCATE_FLUSH or CM_DEALLOCATE_CONFIRM, and the conversation is using an LU 6.2 CRM.
  - The *sync_level* specifies CM_SYNC_POINT_NO_CONFIRM, the *send_receive_mode* is set to CM_HALF_DUPLEX, and the conversation is using an LU 6.2 CRM.
  - The *sync_level* specifies an undefined value.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

If a *return_code* other than CM_OK is returned on the call, the *sync_level* conversation characteristic is unchanged.

Related Information

"Confirm (CMCFM)" on page 133 and "Confirmed (CMCFMD)" on page 137 provide further information on confirmation processing.

"Example 5: Validation of Data Receipt" on page 78 and "Example 15: Sending Program Issues a Commit" on page 98 show how to use the Set_Sync_Level call and how to perform confirm and sync point processing.

"Support for Resource Recovery Interfaces" on page 54 contains additional related information on the Set_Sync_Level call.

Set_TP_Name (CMSTPN)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

Set_TP_Name (CMSTPN) is used by a program that initiates a conversation, using the Initialize_Conversation call, to set the *TP_name* and *TP_name_length* characteristics for a given conversation. Set_TP_Name overrides the current values that were originally acquired from the side information using the *sym_dest_name*. See “Side Information” on page 23 for more information.

This call does not change the values in the side information. Set_TP_Name only changes the *TP_name* and *TP_name_length* characteristics for this conversation. The Set_TP_Name call may also be used by the program that did not initiate the conversation to identify a specific *TP_name* to be accepted on a given conversation.

The *tp_name* conversation characteristic identifies the recipient of the conversation initialization request. It may be set by the conversation initiator.

Note: A program cannot issue Set_TP_Name after an Allocate, Accept_Conversation, or Accept_Incoming is issued.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

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

Note: A program may require special syntax for the name and special authority to specify some TP names. For example, SNA service transaction programs require special authority with LU 6.2. (For more information, see “SNA Service Transaction Programs” on page 727.)

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

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

- CM_OK
- CM_PROGRAM_STATE_CHECK

This value indicates that the conversation is not in **Initialize** or **Initialize_Incoming** state.

- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *TP_name_length* specifies a value less than 1 or greater than 64.
  - The *partner_ID* characteristic is set to a non-null value.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. Specify *TP_name* using the local system's native encoding. CPI Communications automatically converts the *TP_name* from the native encoding where necessary.
2. If a *return_code* other than CM_OK is returned on the call, the *TP_name* and *TP_name_length* conversation characteristics are unchanged.
3. The *TP_name* specified on this call must be formatted according to the naming conventions of the partner system.
4. Refer to "SNA Service Transaction Programs" on page 727 for special handling of SNA Service Transaction Program names.

Related Information

"Side Information" on page 23 and notes 3 on page 652 and 6 on page 653 of Table 61 on page 650 provide further discussion of the *TP_name* conversation characteristic.

"Automatic Conversion of Characteristics" on page 41 provides further information on the automatic conversion of the *TP_name* parameter.

"SNA Service Transaction Programs" on page 727 provides more information on privilege and service transaction programs.

Set_Transaction_Control (CMSTC)

OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32

Set_Transaction_Control (CMSTC) is used by a program to set the *transaction_control* characteristic for a given conversation. Set_Transaction_Control overrides the value that was assigned when the Initialize_Conversation call was issued.

The *transaction_control* conversation characteristic indicates whether the conversation supports chained or unchained transactions. It may be set by the conversation initiator.

Notes:

1. Only the program that initiates the conversation can issue this call.
2. The *transaction_control* characteristic is used only by an OSI TP CRM.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

transaction_control *(input)*
Specifies whether the superior program wants to use chained or unchained transactions on the conversation with the subordinate. The *transaction_control* variable can have one of the following values:

- CM_CHAINED_TRANSACTIONS
  Specifies that the conversation will use chained transactions.
- CM_UNCHAINED_TRANSACTIONS
  Specifies that the conversation will use unchained transactions.

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is not in the **Initialize** state.

- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned identifier.
  - The *transaction_control* specifies an undefined value.
  - The *sync_level* is set to either CM_NONE or CM_CONFIRM.
  - The *transaction_control* specifies CM_UNCHAINED_TRANSACTIONS, and the conversation is using an LU 6.2 CRM.
- CM_OPERATION_NOT_ACCEPTED
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

If a *return_code* other than CM_OK is returned on the call, the *transaction_control* conversation characteristic remains unchanged.

Related Information

"Chained and Unchained Transactions" on page 61 provides more information about using chained and unchained transactions with CPI Communications.

Specify_Local_TP_Name (CMSLTP)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X*					X*			X

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

X* In AIX, this call is supported in Version 3 Release 1 or later. In Communications Manager/2, this call is supported in Communications Server.

Format

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

Parameters

TP_name *(input)*
Specifies a name to be associated with this program.

Note: Refer to “SNA Service Transaction Programs” on page 727 for special handling of SNA Service Transaction Program names.

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

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  – The *TP_name* specifies a name that is restricted in some way by node services.
  – The *TP_name* has incorrect internal syntax as defined by node services.
  – The *TP_name_length* specifies a value less than 1 or greater than 64.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause any state changes.

Usage Notes

1. If a *return_code* other than CM_OK is returned on the call, the names associated with the current program remain unchanged.
2. Any of the names associated with the program at the time an Accept_Conversation or Accept_Incoming call is issued can be used to satisfy the call.
3. If the program has an outstanding Accept_Incoming or Accept_Conversation call when it issues Specify_Local_TP_Name, the names used to satisfy the outstanding Accept_Incoming or Accept_Conversation are not affected. The newly specified name will be added to the names used to satisfy subsequent Accept_Incoming or Accept_Conversation calls.

Related Information

"Accept_Conversation (CMACCP)" on page 119 and "Accept_Incoming (CMACCI)" on page 121 describe how an incoming conversation is accepted by a program.

"Release_Local_TP_Name (CMRLTP)" on page 244 explains additional timing considerations for names associated with the program.

Test_Request_To_Send_Received (CMTRTS)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
X	X	X	X	X	X	X	X	X

Test_Request_To_Send_Received (CMTRTS) is used by a program to determine whether a request-to-send or allocate-confirmed notification has been received from the remote program for the specified conversation.

Note: The Test_Request_To_Send_Received call has meaning only when a half-duplex conversation is being used.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

control_information_received *(output)*
Specifies the variable containing an indication of whether or not control information has been received.

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

- CM_NO_CONTROL_INFO_RECEIVED
  Indicates that no control information was received.
- CM_REQ_TO_SEND_RECEIVED
  The local program received a request-to-send notification from the remote program. The remote program issued Request_To_Send, requesting the local program's end of the conversation to enter **Receive** state, which would place the remote program's end of the conversation in **Send** state. See “Request_To_Send (CMRTS)” on page 246 for further discussion of the local program's possible responses.
- CM_ALLOCATE_CONFIRMED (OSI TP CRM only)
  The local program received confirmation of the remote program's acceptance of the conversation.
- CM_ALLOCATE_CONFIRMED_WITH_DATA (OSI TP CRM only)
  The local program received confirmation of the remote program's acceptance of the conversation. The local program may now issue an Extract_Initialization_Data (CMEID) call to receive the initialization data.
- CM_ALLOCATE_REJECTED_WITH_DATA (OSI TP CRM only)
  The remote program rejected the conversation. The local program may now issue an Extract_Initialization_Data (CMEID) call to receive the initialization data.

This value will be returned with a return code of CM_OK. The program will receive a CM_DEALLOCATED_ABEND return code on a later call on the conversation.

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

Notes:

1. If *return_code* is set to CM_PROGRAM_PARAMETER_CHECK or CM_PROGRAM_STATE_CHECK, the value contained in *control_information_received* has no meaning.

2. When more than one piece of control information is available to be returned to the program, it will be returned in the following order:

   - CM_ALLOCATE_CONFIRMED, CM_ALLOCATE_CONFIRMED_WITH_DATA, or CM_ALLOCATE_REJECTED_WITH_DATA
   - CM_RTS_RCVD_AND_EXP_DATA_AVAIL
   - CM_REQ_TO_SEND_RECEIVED
   - CM_EXPEDITED_DATA_AVAILABLE
   - CM_NO_CONTROL_INFO_RECEIVED

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

- CM_OK
- CM_PROGRAM_STATE_CHECK
  - This value indicates that the conversation is not in **Send**, **Receive**, **Send-Pending**, **Defer-Receive**, or **Defer-Deallocate** state.
  - For a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the conversation's context is in the **Backout-Required** condition. The Test_Request_To_Send_Received call is not allowed for this conversation while its context is in this condition.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *send_receive_mode* of the conversation is CM_FULL_DUPLEX.
- CM_OPERATION_NOT_ACCEPTED
  This value indicates that the program has chosen conversation-level non-blocking for the conversation and a previous call operation is still in progress.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. When the local system receives a request-to-send or allocate-confirmed notification, it retains the notification until the local program issues a call (such as Test_Request_To_Send_Received) with the *control_information_received* parameter. It will retain only one request-to-send or allocate-confirmed notification at a time (per conversation). Additional notifications are discarded until the retained notification is indicated to the local program. Therefore, a remote program may issue the Request_To_Send call more times than are indicated to the local program.

   When the local system receives expedited data from the partner system, it is indicated on the calls that have the *control_information_received* parameter until the expedited data is actually received by the program.

2. After the retained notification, other than the expedited data notification, is indicated to the local program through the *control_information_received* parameter, the local system discards the notification.

3. **Note to Implementers:** A request-to-send or allocate-confirmed notification can be reported on this call (not associated with any queue), on the Send_Expedited_Data call (associated with the Expedited-Send queue), and on the Receive_Expedited_Data call (associated with the Expedited-Receive queue). When the program uses queue-level non-blocking, more than one of these calls may be executed simultaneously. An implementation should report the notification to the program only once, through one of these calls.

4. A program should not rely solely on this call to test whether expedited data is available. Expedited data may be available in the CRM, but the implementation of the CPIC layer may not always be able to indicate this to the program on this call. To test for the availability of expedited data, the program should issue Receive_Expedited_Data with the *expedited_receive_type* set to CM_RECEIVE_IMMEDIATE.

5. On MVS and VM systems, for conversations that cross a VTAM network, the Test_Request_To_Send_Received (CMTRTS) call always sets *request_to_send_received* to CM_REQ_TO_SEND_NOT_RECEIVED when *return_code*=CM_OK, regardless of whether the remote programs have sent such requests to the local programs. For MVS/ESA, CMTRTS works properly in conversations between LUs controlled by APPC/MVS in the same MVS system image. It also works properly under VM within a TSAF or CS collection.

Related Information

"Request_To_Send (CMRTS)" on page 246 provides further discussion of the request-to-send function, and "Set_Allocate_Confirm (CMSAC)" on page 282 provides more information about the allocate-confirmed function.

Wait_For_Completion (CMWCMP)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
				X	X*			X

Wait_For_Completion (CMWCMP) is used to wait for completion of one or more outstanding operations represented in a specified outstanding-operation-ID (OOID) list.

X* In Communications Manager/2, this call is supported in Communications Server.

Format

```
CALL CMWCMP(OOID_list,
            OOID_list_count,
            timeout,
            completed_op_index_list,
            completed_op_count,
            user_field_list,
            return_code)
```

Parameters

OOID_list *(input)*
Specifies a list of OOIDs representing the outstanding operations for which completion is expected.

OOID_list_count *(input)*
Specifies the number of OOIDs contained in *OOID_list*.

timeout *(input)*
Specifies the amount of time in milliseconds that the program is willing to wait for completion of an operation. Valid *timeout* values are zero or any greater integer number.

completed_op_index_list *(output)*
Specifies a list of indexes corresponding to the OOIDs in *OOID_list* for which the associated operations have completed. The index is the position of an OOID in *OOID_list*, beginning with 1.

completed_op_count *(output)*
Specifies the number of indexes contained in *completed_op_index_list*, or the number of user fields contained in *user_field_list*, or both.

user_field_list *(output)*
Specifies a list of user fields corresponding to the completed operations.

return_code *(output)*

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

- CM_OK
- CM_CALL_NOT_SUPPORTED
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  - The *OOID_list_count* specifies a value less than 1.
  - The number of OOIDs in *OOID_list* is less than the value specified in *OOID_list_count*.
  - The *OOID_list* contains an unassigned OOID.
  - The *timeout* specifies a value less than zero.
- CM_PROGRAM_STATE_CHECK
  This value indicates that there is no outstanding operation associated with any of the OOIDs specified in *OOID_list*.
- CM_UNSUCCESSFUL
  This value indicates that the specified timeout value has elapsed and none of the operations specified in *OOID_list* has completed.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

This call does not cause a state change.

Usage Notes

1. Unless the *return_code* indicates CM_OK, the values of all other parameters on this call have no meaning.
2. The call returns the OOID corresponding to a completed operation only once. At that time, all information about the completed operation is purged from the associated queue. If the program issues the call to check the status of the same operation again, the OOID will not be returned.
3. When the call returns a completion operation to the program, the return code of the completion operation can be found in the *return_code* parameter on the completed call.
4. A special timeout value of zero can be used to check the status of all the operations whose OOIDs are specified in the *OOID_list* parameter. The call specified in this way incurs no blocking.
5. The program can replace a previously returned OOID in the *OOID_list* parameter with a null OOID (integer zero) and continue to use the same list for the following Wait_For_Completion call. The null OOID is not associated with any outstanding operation.
6. There is a one-to-one correspondence between elements of the completed-operation-index list and those of the user-field list. Hence, the size of the user-field list equals that of the completed-operation-index list.
7. The program should allocate the same amount of storage for the completed-operation-index list and user-field list as it does for the outstanding-operation-ID list. If there is not enough storage allocated, the program may lose some OOIDs and user fields that correspond to the completed operations.

8. In a multi-threaded environment, concurrent Wait_For_Completion operations can occur. If an OOID is specified on more than one Wait_For_Completion call, the OOID is returned on only one of the Wait_For_Completion calls when the corresponding outstanding operation completes.
9. **Note to Implementers:** After returning CM_OPERATION_INCOMPLETE to the program, an implementation should not fill the return code for the outstanding operation before the program checks the return-code value. It is recommended that implementations fill the return code only when the program issues a Wait_For_Completion call for the outstanding operation.

Related Information

"Non-Blocking Operations" on page 47 discusses the use of non-blocking operations.

"Example 13: Accepting Multiple Conversations Using Conversation-Level Non-Blocking Calls" on page 94 shows an example of a program that uses conversation-level non-blocking calls to accept multiple incoming half-duplex conversations.

"Set_Queue_Processing_Mode (CMSQPM)" on page 340 describes how to set the processing mode for a conversation queue.

Wait_For_Conversation (CMWAIT)

LU 6.2	OSI TP

AIX	CICS	IMS	MVS	NS/WIN	OS/2	OS/400	VM	WIN32
					X*			X

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

X* In Communications Manager/2, this call is supported in Communications Server.

Format

CALL CMWAIT(*conversation_ID*,
conversation_return_code,
return_code)

Parameters

conversation_ID *(output)*
Specifies the variable containing the conversation identifier for the completed operation.

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

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

- CM_OK
- CM_ALLOCATE_FAILURE_NO_RETRY
- CM_ALLOCATE_FAILURE_RETRY
- CM_BUFFER_TOO_SMALL
- CM_CALL_NOT_SUPPORTED
- CM_CONV_DEALLOC_AFTER_SYNCPT
- CM_CONVERSATION_CANCELLED
- CM_CONVERSATION_ENDING
- CM_CONVERSATION_TYPE_MISMATCH
- CM_DEALLOC_CONFIRM_REJECT
- CM_DEALLOCATED_ABEND
- CM_DEALLOCATED_ABEND_BO

- CM_DEALLOCATED_ABEND_SVC
- CM_DEALLOCATED_ABEND_SVC_BO
- CM_DEALLOCATED_ABEND_TIMER
- CM_DEALLOCATED_ABEND_TIMER_BO
- CM_DEALLOCATED_NORMAL
- CM_DEALLOCATED_NORMAL_BO
- CM_EXP_DATA_NOT_SUPPORTED
- CM_INCLUDE_PARTNER_REJECT_BO
- CM_PIP_NOT_SPECIFIED_CORRECTLY
- CM_PRODUCT_SPECIFIC_ERROR
- CM_PROGRAM_ERROR_NO_TRUNC
- CM_PROGRAM_ERROR_PURGING
- CM_PROGRAM_ERROR_TRUNC
- CM_RESOURCE_FAIL_NO_RETRY_BO
- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAILURE_RETRY
- CM_RESOURCE_FAILURE_RETRY_BO
- CM_RETRY_LIMIT_EXCEEDED
- CM_SECURITY_MUTUAL_FAILED
- CM_SECURITY_NOT_SUPPORTED
- CM_SECURITY_NOT_VALID
- CM_SEND_RCV_MODE_NOT_SUPPORTED
- CM_SVC_ERROR_NO_TRUNC
- CM_SVC_ERROR_PURGING
- CM_SVC_ERROR_TRUNC
- CM_SYNC_LVL_NOT_SUPPORTED_PGM
- CM_SYNC_LVL_NOT_SUPPORTED_SYS
- CM_TAKE_BACKOUT
- CM_TP_NOT_AVAILABLE_NO_RETRY
- CM_TP_NOT_AVAILABLE_RETRY
- CM_TPN_NOT_RECOGNIZED

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

return_code *(output)*

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

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

State Changes

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

Usage Notes

1. Wait_For_Conversation waits for the completion of any outstanding operation on any conversation using conversation-level non-blocking. It is the responsibility of the program to keep track of the operation in progress on each conversation in order to be able to interpret properly the *conversation_return_code* value.
2. In a multi-threaded environment, concurrent operations may occur. A Wait_For_Conversation call waits for any operation, on any conversation using conversation-level non-blocking, that either is already outstanding or becomes outstanding during execution of the Wait_For_Conversation call. In case of concurrent Wait_For_Conversation operations, completion of an outstanding operation is indicated on one Wait_For_Conversation call only.
3. It is the responsibility of the event-handling portion of the program to record sufficient information for the program to decide how to proceed on receipt of the CM_SYSTEM_EVENT return code.
4. The program's current context is not changed as a result of the completion of a Wait_For_Conversation call.
5. This call applies only to conversations using conversation-level non-blocking support.

Related Information

"Non-Blocking Operations" on page 47 discusses the use of non-blocking operations.

"Example 13: Accepting Multiple Conversations Using Conversation-Level Non-Blocking Calls" on page 94 shows an example of a program that uses conversation-level non-blocking calls to accept multiple incoming half-duplex conversations.

"Cancel_Conversation (CMCANC)" on page 131 describes the means for terminating an operation before it is completed.

"Set_Processing_Mode (CMSPM)" on page 334 describes setting the *processing_mode* conversation characteristic.

Part 3. CPI-C 2.1 Implementation Specifics

Chapter 5. CPI Communications on AIX 381
AIX Publications 381
AIX Operating Environment 382
  AIX CPI Communications Concepts 382
  Conformance Classes Supported 382
  Languages Supported 383
  Pseudonym Files 383
  Profiles 383
  Creating and Maintaining Profiles through SMIT 388
  Starting SMIT 388
  Working with Profiles 388
  Verifying Profiles 388
  How Dangling Conversations Are Deallocated 388
  Scope of the Conversation_ID 389
  Identifying Product-Specific Errors 389
  Diagnosing Errors 389
  When Allocation Requests Are Sent 391
  Deviations from the CPI Communications Architecture 391
  Security Using CPI Communications and AIX 391
  Compilation 393
  Running a Transaction Program 393
AIX Extension Calls 394
Extract_Conversation_Security_Type (XCECST) 396
Extract_Conversation_Security_User_ID (XCECSU) 398
Set_Conversation_Security_Password (XCSCSP) 399
Set_Conversation_Security_Type (XCSCST) 401
Set_Conversation_Security_User_ID (XCSCSU) 403
Set_Signal_Behavior (XCSSB) 405

Chapter 6. CPI Communications on CICS/ESA 407
CICS/ESA Publications 407
CICS/ESA Operating Environment 407
  Conformance Classes Supported 408
  Languages Supported 408
  Pseudonym Files 408
  Defining Side Information 409
  How Dangling Conversations Are Deallocated 411
  Scope of the Conversation_ID 412
  Identifying Product-Specific Errors 412
  Diagnosing Errors 412
  When Allocation Requests are Sent 412
  Deviations from the CPI Communications Architecture 413
CICS/ESA Extension Calls 413
CICS/ESA Special Notes 414

Chapter 7. CPI Communications on IMS/ESA 415

Chapter 8. CPI Communications on MVS/ESA 417
MVS/ESA Publications 417
MVS/ESA Operating Environment 418

© Copyright IBM Corp. 1996, 1998

Conformance Classes Supported . . . 418
Languages Supported . . . 419
Pseudonym Files . . . 419
Defining Side Information . . . 420
How Dangling Conversations Are Deallocated . . . 420
Scope of the Conversation_ID . . . 420
Identifying Product-Specific Errors . . . 421
Diagnosing Errors . . . 423
When Allocation Requests Are Sent . . . 424
Deviations from the CPI Communications Architecture . . . 424
MVS/ESA Extension Calls . . . 425
MVS/ESA Special Notes . . . 425
TP Profiles . . . 425
MVS Performance Considerations . . . 425
APPC/MVS Services . . . 425

Chapter 9. CPI Communications on Networking Services for Windows 429
Networking Services for Windows Publications . . . 429
Networking Services for Windows Operating Environment . . . 429
Support of CPI-C Conformance Classes . . . 429
Optional Conformance Classes Supported . . . 429
Optional Conformance Classes Not Supported . . . 430
Languages Supported . . . 430
Pseudonym Files . . . 430
Examples of Using C . . . 431
CPI-C Function Calls in C . . . 431
Using the Pseudonym Files in C Language Programs . . . 431
Using Other Languages . . . 431
Linking with the CPI-C Import Library . . . 432
Memory Considerations . . . 432
Data Buffers . . . 432
Stack Size . . . 432
Defining Side Information . . . 432
Usage Notes for Mode_Name and TP_Name . . . 432
Mode_Name . . . 432
Restrictions on Transaction Program Names . . . 433
How Dangling Conversations Are Deallocated . . . 433
Diagnosing Errors . . . 433
Log_Data . . . 433
Identifying Product-Specific Errors . . . 433
Deviations from the CPI Communications Architecture . . . 434
Return_control Characteristic for Allocate (CMALLC) . . . 434
CM_PROGRAM_PARAMETER_CHECK Return Code . . . 434
Log Data Support . . . 434

Chapter 10. CPI Communications on OS/2 . . . 435
OS/2 Publications . . . 436
OS/2 Operating Environment . . . 437
Conformance Classes Supported . . . 437
Languages Supported . . . 437
C . . . 438
COBOL . . . 439
FORTRAN . . . 439
REXX (SAA Procedures Language) . . . 440

Pseudonym Files . . . 443
Defining Side Information . . . 443
User-Defined Side Information . . . 444
Program-Defined Side Information . . . 444
Side Information Parameters . . . 445
How Dangling Conversations Are Deallocated . . . 447
Scope of the Conversation_ID . . . 447
Identifying Product-Specific Errors . . . 447
Diagnosing Errors . . . 448
Set_Log_Data (CMSLD) . . . 448
Logging Errors for CPI Communications Error Return Codes . . . 448
Causes for the CM_PROGRAM_PARAMETER_CHECK Return Code . . . 449
Causes for the CM_PROGRAM_STATE_CHECK Return Code . . . 449
When Allocation Requests Are Sent . . . 450
Deviations from the CPI Communications Architecture . . . 450
Accept_Incoming (CMACCI) . . . 450
Release_Local_TP_Name (CMRLTP) . . . 451
Specify_Local_TP_Name (CMSLTP) . . . 451
Set_Sync_Level (CMSSL) . . . 451
Programming Languages Not Supported . . . 452
Mode Names Not Supported . . . 452
CPI Communications Functions Not Available . . . 452
OS/2 Extension Calls—System Management . . . 454
Delete_CPIC_Side_Information (XCMDSI) . . . 455
Extract_CPIC_Side_Information (XCMESI) . . . 457
Set_CPIC_Side_Information (XCMSSI) . . . 460
Define_TP (XCDEFTP) . . . 463
Delete_TP (XCDELTP) . . . 467
Register_Memory_Object (XCRMO) . . . 469
Unregister_Memory_Object (XCURMO) . . . 470
OS/2 Extension Calls—Conversation . . . 471
Extract_Conversation_Security_Type (XCECST) . . . 472
Extract_Conversation_Security_User_ID (XCECSU) . . . 473
Initialize_Conv_For_TP (XCINCT) . . . 474
Set_Conversation_Security_Password (XCSCSP) . . . 476
Set_Conversation_Security_Type (XCSCST) . . . 477
Set_Conversation_Security_User_ID (XCSCSU) . . . 478
OS/2 Extension Calls—Transaction Program Control . . . 479
End_TP (XCENDT) . . . 480
Extract_TP_ID (XCETI) . . . 482
Start_TP (XCSTP) . . . 483
OS/2 Special Notes . . . 485
Migration to Communications Server . . . 485
Multi-threaded CPI-C Programs . . . 485
Considerations for CPI Communications Calls . . . 485
TP Instances for Communications Manager . . . 486
Accept_Conversation (CMACCP) or Accept_Incoming (CMACCI) . . . 487
Extract_Conversation_Context (CMECTX) . . . 488
Extract_Secondary_Information (CMESI) . . . 489
Initialize_Conversation (CMINIT) . . . 490
Receive (CMRCV) . . . 492
Send_Data (CMSEND) . . . 492
Send_Expedited_Data (CMSNDX) . . . 493
Set_Partner_LU_Name (CMSPLN) . . . 493

Set_Sync_Level (CMSSL) 493
Set_Queue_Processing_Mode (CMSQPM) 493
Test_Request_To_Send (CMTRTS) 493
Wait_For_Completion (CMWCMP) 493
Characteristics, Fields, and Variables 494
Communications Manager Native Encoding 494
Variable Types and Lengths 495
Defining and Running a CPI Communications Program on Communications Manager 498
Defining a CPI Communications Program to Communications Manager 498
Using Defaults for TP Definitions 498
Communications Manager Use of OS/2 Environment Variables 499
Stack Size 500
Performance Considerations For Using Send/Receive Buffers 500
Exit List Processing 501
Sample Program Listings for OS/2 502
OS/2 C Sample Programs 503
SETSIDE.C 503
OS/2 COBOL Sample Programs 504
DEFSIDE.CBL 504
DELSIDE.CBL 507
OS/2 REXX Sample Programs 509
XCMSSI.CMD 509
XCMESI.CMD 510

Chapter 11. CPI Communications on Operating System/400 513
OS/400 Publications 513
OS/400 Operating Environment 513
OS/400 Terms and Concepts 513
Conformance Classes Supported 515
Languages Supported 515
Pseudonym Files 516
Defining Side Information 516
Managing the Communications Side Information 517
How Dangling Conversations Are Deallocated 518
Reclaim Resource Processing 519
Scope of the Conversation_ID 519
Identifying Product-Specific Errors 519
CM_PRODUCT_SPECIFIC_ERROR 519
Diagnosing Errors 520
OS/400 CPI Communications Support of Log_Data 521
Return Codes 521
REXX Reserved RC Variable 522
REXX Error and Failure Conditions 523
Tracing CPI Communications 523
When Allocation Requests Are Sent 524
OS/400 Extension Calls 524
OS/400 Special Notes 524
CPI Communications over TCP/IP Support 524
Prestarting Jobs for Incoming Conversations 524
Multiple Conversation Support 525
Portability Considerations 525

Chapter 12. CPI Communications on VM/ESA CMS 527

VM Publications . . . 527
VM/ESA Operating Environment . . . 528
Conformance Classes Supported . . . 528
Languages Supported . . . 528
Programming Language Considerations . . . 529
Pseudonym Files . . . 530
Defining Side Information . . . 532
How Dangling Conversations Are Deallocated . . . 534
Scope of the Conversation_ID . . . 534
Identifying Product-Specific Errors . . . 534
Diagnosing Errors . . . 536
Processing Log Data . . . 536
Invocation Errors . . . 537
Possible Causes for Selected Return Codes . . . 538
APPC Protocol Errors in VM/ESA . . . 539
When Allocation Requests Are Sent . . . 540
Deviations from the CPI Communications Architecture . . . 540
VM/ESA Extension Calls . . . 541
Extract_Conversation_LUWID (XCECL) . . . 544
Extract_Conversation_Security_User_ID (XCECSU) . . . 546
Extract_Conversation_Workunitid (XCECWU) . . . 548
Extract_Local_Fully_Qualified_LU_Name (XCELFQ) . . . 550
Extract_Remote_Fully_Qualified_LU_Name (XCERFQ) . . . 552
Extract_TP_Name (XCETPN) . . . 554
Identify_Resource_Manager (XCIDRM) . . . 555
Set_Client_Security_User_ID (XCSCUI) . . . 559
Set_Conversation_Security_Password (XCSCSP) . . . 562
Set_Conversation_Security_Type (XCSCST) . . . 564
Set_Conversation_Security_User_ID (XCSCSU) . . . 566
Signal_User_Event (XCSUE) . . . 568
Terminate_Resource_Manager (XCTRRM) . . . 570
Wait_on_Event (XCWOE) . . . 571
VM/ESA Variables and Characteristics . . . 576
Pseudonyms and Integer Values . . . 576
Variable Types and Lengths . . . 577
VM/ESA Special Notes . . . 577
Program-Startup Processing . . . 578
End-of-Command Processing . . . 578
Work Units . . . 578
External Interrupts . . . 579
Coordination with the SAA Resource Recovery Interface . . . 579
Additional Conversation Characteristics . . . 579
TP-Model Applications in VM/ESA . . . 580
LU 6.2 Communications Model . . . 580
VM/ESA TP-Model Applications . . . 581
Implications . . . 582
VM/ESA-Specific Notes for CPI Communications Routines . . . 583
VM/ESA Communications Events . . . 585
The VMCPIC Event . . . 586
Notes on the VMCPIC Event . . . 587
Using the Online HELP Facility . . . 588

Chapter 13. CPI Communications on IBM eNetwork Personal Communications V4.1 for Windows 95 . . . 589

Conformance Classes Supported . . . 589
Personal Communications V4.1 for Windows 95 Publications . . . 590
Programming Language Support . . . 590
Linking with the CPI-C library . . . 590
Accepting Conversations . . . 590
Extension Calls supported . . . 591
WinCPICStartup . . . 591
WinCPICCleanup . . . 591
Specify_Windows_Handle (XCHWND) . . . 591
Deviations from the CPI-C architecture . . . 592

Chapter 14. CPI Communications on Win32 and 32-bit API Client Platforms . . . 593
Operating Environment . . . 594
Conformance Classes Supported . . . 594
Languages Supported . . . 596
CPI-C Communications Use of Environment Variables . . . 597
Pseudonym Files . . . 597
Defining Side Information . . . 598
User-Defined Side Information . . . 598
Program-Defined Side Information . . . 598
How Dangling Conversations Are Deallocated . . . 599
Diagnosing Errors . . . 599
Causes for the CM_PROGRAM_PARAMETER_CHECK Return Code . . . 599
Causes for the CM_PROGRAM_STATE_CHECK Return Code . . . 599
When Allocation Requests Are Sent . . . 600
Deviations from the CPI Communications Architecture . . . 600
Accept_Incoming (CMACCI) . . . 600
Release_Local_TP_Name (CMRLTP) . . . 600
Mode Names Not Supported . . . 600
CPI-C Communication Functions Not Available . . . 600
Extension Calls – System Management . . . 601
Delete_CPIC_Side_Information (XCMDSI) . . . 602
Extract_CPIC_Side_Information . . . 604
Set_CPIC_Side_Information . . . 607
Extension Calls—Conversation . . . 610
Extract_Conversation_Security_Type (XCECST) . . . 611
Extract_Conversation_Security_User_ID (XCECSU) . . . 612
Initialize_Conv_For_TP (XCINCT) . . . 613
Set_Conversation_Security_Password (XCSCSP) . . . 615
Set_Conversation_Security_Type (XCSCST) . . . 616
Set_Conversation_Security_User_ID (XCSCSU) . . . 617
Extension Calls—Transaction Program Control . . . 618
End_TP (XCENDT) . . . 619
Extract_TP_ID (XCETI) . . . 621
Start_TP (XCSTP) . . . 622
Special Notes . . . 624
Migration to Communications Server . . . 624
Multi-threaded CPI-C Programs . . . 624
Considerations for CPI Communications Calls . . . 624
TP Instances for CPI-C Communications . . . 625
Accept_Conversation (CMACCP) or Accept_Incoming (CMACCI) . . . 625
Extract_Conversation_Context (CMECTX) . . . 627
Extract_Secondary_Information (CMESI) . . . 627

Initialize_Conversation (CMINIT) 628
Receive (CMRCV) 629
Send_Data (CMSEND) 630
Send_Expedited_Data (CMSNDX) 630
Set_Partner_LU_Name (CMSPLN) 630
Set_Queue_Processing_Mode (CMSQPM) 630
Test_Request_To_Send (CMTRTS) 631
Wait_For_Completion (CMWCMP) 631
Characteristics, Fields, and Variables 631
Variable Types and Lengths 632
WOSA Extension Calls Supported 633
WinCPICStartup 634
Returns 634
WinCPICCleanup 634
WINCPICCleanup() 634
Specify_Windows Handle (xchwnd) 635

Chapter 5. CPI Communications on AIX

This chapter summarizes the product-specific information that the AIX programmer needs when writing application programs that contain CPI Communications calls or AIX extension calls. This information is for use with the CPI-C included in AIX SNA Server/6000, Version 3, Release 1, Version 2, Release 1.2, or Desktop SNA for AIX, Version 1.

AIX SNA Server/6000 (hereafter referred to as SNA Server/6000) runs on all models of the IBM RISC System/6000 workstation. SNA Server/6000 supports application program use of CPI Communications calls to communicate with programs on other RISC System/6000 workstations or on other systems in an SNA network. Communication with other programs on the same RISC System/6000 workstation is supported in a limited fashion, as described under "AIX Operating Environment" on page 382.

The implementations of CPI Communications calls by SNA Server/6000 are compatible with those of other systems documented in this book. This chapter describes aspects of CPI Communications that are unique to SNA Server/6000.

This chapter is organized as follows:

- AIX Publications
- AIX Operating Environment
  - Conformance Classes Supported
  - Languages Supported
  - Pseudonym Files
  - Profiles
  - How Dangling Conversations Are Deallocated
  - Scope of the Conversation_ID
  - Identifying Product-Specific Errors
  - Diagnosing Errors
  - When Allocation Requests Are Sent
  - Deviations from the CPI Communications Architecture
  - Security Using CPI Communications and AIX
  - Compilation
  - Running a Transaction Program
- AIX Extension Calls

AIX Publications

The following AIX publications contain detailed product information:

- *AIX SNA Server/6000 User's Guide*, SC31-7002
- *AIX SNA Server/6000 Transaction Program Reference*, SC31-7003
- *AIX SNA Server/6000 Command Reference*, SC31-7100
- *AIX SNA Server/6000 Configuration Reference*, SC31-7014
- *Introduction to CPI-C Programming in an AIX SNA Environment*, GG22-9510
- *SNA Server for AIX: Version 3.1 User's Guide* SC31-8211-00
- *SNA Server for AIX: Version 3.1 Transaction Program Reference* , SC31-8212-00
- *SNA Server for AIX: Version 3.1 Command Reference*, SC31-8214-00

© Copyright IBM Corp. 1996, 1998

- *SNA Server for AIX: Version 3.1 Configuration Reference*, SC31-8213-0
- *SNA Server for AIX: Version 3.1 CPI-C Programming Guide*, GC31-8210-00

AIX Operating Environment

On the RISC System/6000 workstation, programs that use CPI Communications calls are considered *transaction programs* (TPs). After reading this chapter, refer to *AIX SNA Server/6000: Transaction Program Reference* for details on designing, writing, testing, and installing transaction programs to run on AIX. Refer to *AIX SNA Server/6000: Configuration Reference* for details on how to supply side information and how and when to create TP profiles for transaction programs on AIX.

The CPI Communications calls are part of the libraries **libcpic.a** and **libcpic_r.a**, which is shipped with SNA Server/6000. These calls are structured to act as independent C-language calls. SNA Server/6000 offers some extensions to the basic CPI Communications interface. These extension calls have a prefix of **xc**, rather than the **cm** prefix used for the basic CPI Communications calls.

AIX CPI Communications Concepts

In AIX, a source CPI Communications program is run under the user ID the user is operating under. A target program is run under the user ID specified in the transaction program profile.

Allocations using the already verified option will always use the source program's user ID. If the user making an allocation request is part of the system group, the user ID for the allocation request can be changed with a Set_Security_User_ID call.

The following sections explain some special considerations that should be understood when writing applications for an AIX environment. For more information, refer to *Introduction to CPI-C Programming in an AIX SNA Environment*.

Conformance Classes Supported

AIX supports the following conformance classes:

- Conversations

  All conversations, with the exception of Extract_Maximum_Buffer_Size (CMEMBS)

- LU 6.2

SNA Server/6000 Version 3 release 1 supports the following additional conformance classes

- Conversations

  Including Extract_Maximum_Buffer Size

- Server
- Security

  except that the security type CM_SECURITY_PROGRAM_STRONG is not supported

- Data Conversion Routines

Refer to "Functional Conformance Class Descriptions" on page 746 for a complete description of functional conformance classes.

Languages Supported

The C language is the only language that can be used to call CPI Communications routines and AIX extension routines.

Pseudonym Files

The pseudonym file for CPI Communications calls in the C language is **/usr/include/cmc.h**. All function prototypes and constants used by the SNA Server/6000 CPI Communications subroutines are defined in this file.

Profiles

To run CPI Communications on SNA Server/6000, a set of physical profiles related to a link station profile and a subset of LU 6.2 session profiles are required.

When generating profiles, keep in mind that the profile name itself is merely a label on the local AIX system. It has no relationship to any name on the remote system. However, the names defined inside these profiles do have a relationship to the remote system. For example, the TP profile name is only referenced in the profile database. But the TP name within the TP profile is the name used to determine which TP is started by a remote allocation request.

Side information for AIX CPI Communications is contained in *side information profiles*. These profiles are created as part of the SNA Server/6000 configuration process, using the System Management Interface Tool (SMIT).

Side_information profiles include a mode name, a partner LU name, and a remote transaction program name.

It helps to understand what profiles are needed and how they relate to each other. Below is a listing of all AIX SNA profiles. All can be entered or changed using various command-line commands (mksnaobj, chsnaobj) or through SMIT.

- Control Point Profile

  This defines the node name the local machine uses for all transaction programs. You should use other manuals to best determine characteristics. But for CPI-C purposes, you should understand that there is only one local node defined on a machine. Configuration is easiest when the network name is common to all machines that you want to establish links to, and AIX SNA can use the control point name as a default LU name. Fully qualified LU names are in the form "Network_Name.LU_Name."

- SNA Profile

  This defines characteristics of the local node. Again, there is only one such profile on the local machine. The important fields include the maximum number of sessions and conversations that can occur on the local machine and whether or not inbound partner LU definitions are allowed (should be set to `yes`).

- SNA DLC Profile

  These profiles fall under the link profile definitions and are specific to the type of physical link you are using. You should use other manuals to help fine-tune

your physical link characteristics. In most cases, the default values should provide acceptable results.

- Link Station Profile

  These profiles, also under the link profile definitions, provide link_station access to the physical links. Important fields include the SNA DLC Profile name, which refers to the name of the DLC profile you are using for physical access.

  The method of link access is also determined through this profile. If you are using this profile as a calling link, you need to determine whether you are calling by link_name or link_address (the network address of the remote machine's physical link—usually obtained through the netstat -v command).

  The link activation parameters may be a source of error. Calling link stations should set *Initiate call when link station is activated* to `yes` and *activate link station at SNA Server startup* to `no`. Listening link stations should have both fields reversed. For both listening and calling link stations, set the activate on demand field to `no`.

  All attachments between two machines consist of a calling link station and a listening link station. Whether a program runs on a calling side or a listening side has absolutely nothing to do with whether a program can be a source or a target. Once the link station is up, you can initialize from either side.

Note: The following profiles all are part of the LU 6.2 session category. Aside from the mode profile on both systems and the transaction program name profile on the target system, none of these profiles are absolutely required. Side information profiles are recommended for the source system, and session security, local LU and partner LU profiles are used with conversation security.

- Local LU Profile

  This profile defines a local LU to the network. The local LU name and alias fields are used to refer to the local LU name. An alias allows you to refer to an LU in a more mnemonic fashion if the regular LUs on a system have hard-to-remember names. An alias is a nick-name.

  If you are using dependent LU 6.2 profiles, you can set the LU address through this profile. This profile is required for dependent LUs.

  If you are using security, the Conversation Security Access List Profile Name allows you to specify the list that contains user names that can start a dynamic transaction program on this LU.

- CPI-C Side Information Profile

  This profile is used by source programs to define names and profiles used when starting a transaction program. All this profile does is provide SNA another way to fill in information for an allocate call in place of the individual CPI-C set calls.

  Relevant fields:

    Local LU or Control Point Alias—name of local LU being accessed by the transaction program. If this field is left blank, then the control point is used as a default.

    (Fully qualified) Partner LU Name—name of remote LU. This can be an alias of a fully-qualified name. It is a field defined for both fully-qualified names and aliases. Only one of these two fields should be used.

Mode Name—name of the mode you want to use for the session between these two programs.

Remote Transaction Program Name—transaction program name on the remote machine, which references the name of the target program executable you want to start using this local program.

RTPN in Hexadecimal—whether or not the remote transaction program name is specified in hexadecimal form. Service TPs, for example, are referenced by hexadecimal numbers.

- Partner LU Profile

  This profile contains information about a remote LU that is a potential partner for sessions containing conversations.

Relevant Fields:

Fully-qualified Partner LU Name—fully-qualified name of the remote LU.

Partner LU Alias—nickname the local LU can use to refer to the partner LU.

Parallel Sessions Supported—whether or not you will accept more that one session from the remote LU. This field is set to `no` with dependent LU 6.2, and to `yes` only with independent LU 6.2 when you want to have more than one session active concurrently.

Session Security Supported—whether or not you require session-level security for sessions with the remote LU. This is related to BIND passwords and not to conversation-level security.

Conversation Security Level—amount of security information the remote LU is allowed to send in an allocation request.

- Mode Profile

  This profile defines an SNA mode, which defines certain attributes for a session. Unless you have very specific transmission needs, you should be able to use one of the pre-defined modes included with AIX SNA. As far as CPI-C is concerned, you want to ensure the maximum number of sessions is large enough to handle the number of programs you want to run using the link, and you might want to control the Maximum RU Size, which is the maximum acceptable size of the send buffer (size negotiated when the bind takes place). The mode name must match the mode name defined on the remote system.

- Transaction Program Name Profile—contains information about local transaction programs. This profile is optional (and not referenced) for source programs, but is required for target programs.

  Relevant Fields:

  Transaction Program Name—name of the program you want to use (should match the remote TP name being sent by the source TP. Unless the *TPN in Hex* field is set to `no`, this field will be translated to EBCDIC before storing this profile).

  TPN in Hexadecimal?—Set this value to `no` unless the name in the above field is in hexadecimal form.

  PIP Data—CPI-C does not use PIP data.

  Conversation Type—conversation type used by the source TP. Use “either” if you do not want to limit remote access to one conversation type.

  Sync Level—sync level used by the source TP. Use “none/confirm” in this field if you do not want to limit remote access to one synchronization level.

  Resource Security Level—amount of security information required in an allocate request to start the transaction program referenced by this profile.

  Resource Security Access List Profile—list which, if defined, contains user IDs that are allowed to remotely start this program. This field is optional and only relevant when the resource security level is set to “access.”

  Full Path to TPN Executable—fully-qualified name (and directory, starting from root) where the executable is located (example: **/u/cpictest/target**).

  User ID—ID number you wish to execute the program under. This value can produce confusing errors in the source program when it attempts to

allocate. Make sure the user ID you use has permission to run the executable specified in the full path field above. The user referenced by the ID may need read, write or execute permission on directories on the machine, depending on non-CPI-C calls within the program.

Standard Output File/Device—usually, a file to which output from the target program is written during execution. If this is left at **/dev/console**, output will appear on the console. You can change the console by issuing the **swcons** command.

Standard Error File/Device—same as above.

Security Required—level of incoming security required to run this program.

You can use the default values for all other fields.

- Conversation Security Access List (under Conversation Security)—this profile is optional. It contains a list of user IDs (by name) that are allowed to allocate sessions using the local LU profile that refers to this list.
- Resource Security Access List (under Conversation Security)—this profile is optional. It contains a list of user IDs (by name) that are allowed to start the target program that refers to this list (in its TP Name profile).
- Partner LU 6.2 Location – This is needed if the machine is not in an APPN network. It defines where LUs can be found on the network.

  Relevant fields:

  Fully-qualified Partner LU Name—The LU name entered must be the partner LU name as it is defined on the remote system. Full and partial wildcards are supported. A full wildcard is useful for an APPN network node attached to a LEN-level subarea network.

  Partner LU Location Method—Two possible values:

  - Owning CP—The partner LU will be located by first finding the control point specified in the "fully-qualified control point name" field.
  - Link station—The partner LU is assumed to be located on the other side of the link specified by the "Link Station Profile Name" field.

    Note: The link station option is only valid if the local machine is not defined as a network node.

  Fully-qualified Owning CP Name—Network name of the machine where the LU is located.

  Local Node is Network Server for LEN Node—Specifying "yes" is only valid if the local machine is an APPN network node.

  Local LU Name—Specify what local LU name to use over this link.

  Link Station Profile Name—This parameter specifies the link over which the session identified by the "fully-qualified partner LU name" and "local LU name" fields is to activate.

Creating and Maintaining Profiles through SMIT

You can create and maintain profiles through the System Management Interface Tool (SMIT) and through direct profile management commands. SMIT is recommended for manipulating profiles. The following paragraphs describe how to use SMIT with SNA Server/6000.

Starting SMIT

To start SMIT for the SNA Server, type `smit sna` from a command line. If you want to avoid the AIX windows motif version, type `smit -C sna`. To move around in SMIT, use the arrow keys to change the highlighted field and the return key to select the highlighted field.

Working with Profiles

To work with profiles, select “Configure SNA Profiles” from the first screen. To create profiles using quick_config, select the “Quick Configuration” option. To manipulate or create profiles, select “Advanced Configuration.”

Select “Links” from Advanced Configuration to manipulate or create link_station profiles. (The Control Point Profile and the SNA System profile are under the “Advanced Configuration” menu, and the LU6.2 profiles are under “Sessions.” This screen also contains the options for saving and importing profiles.)

To save a profile set, select “Export Configuration Profiles.” This creates a file that contains every profile currently in the SNA database. To take every profile from one of these files and put it in the current database, select “Import Configuration Profiles.” It is a good idea to export profiles after you have created a working configuration.

Verifying Profiles

Use the “Verify Configuration Profiles” option to check required cross references between profiles and enter the profiles into the working set of SNA profiles. (Running this option can help you uncover mistakes.)

Once you are familiar with SMIT screens, you can find, change, and create profiles. When you have a long list of profiles under a certain type and do not remember the specific names, use the F4 key to list them.

After you select a particular profile (or created a new profile), press the `Enter` key to enter the changes or enter the new profile in the database. To cancel the changes, press the F3 key instead of the Enter key.

If you change or create a profile, you will need to verify the profile set to activate the changes. In some cases, you will have to stop a link_station or SNA before you can use your new or changed version.

How Dangling Conversations Are Deallocated

Programs should deallocate conversations before ending, and should have recovery routines to deallocate conversations in the event of abnormal termination. If a local or remote program ends without proper deallocation processing (that is, without sending or receiving a deallocation notification) SNA Server/6000 issues the equivalent of a DEALLOCATE_ABEND return code. The partner program receives a CM_DEALLOCATED_ABEND return code for that condition.

Scope of the Conversation_ID

In AIX, all CPI Communications calls for a given conversation must be made from the same process. The conversation ID is not valid outside the process that initialized or accepted it.

Identifying Product-Specific Errors

The return code CM_PRODUCT_SPECIFIC_ERROR in SNA Server/6000 means that an SNA error code has been stored in the global variable **errno**. SNA error codes are defined and described in the file **/usr/include/luxsna.h**.

Diagnosing Errors

This section helps you debug and understand what can go wrong with transaction programs. You might benefit from setting up CPI-C traces. You can set up tracing on either a source program or a target program (or both). To set up application-level traces, enter the following from a command line:

sna -setlogs -s

This only needs to be done once after installation. It tells SNA to put all trace information in a file in the directory **/var/sna** called **snaservice.X**, where **X** is a number from 1 to 10.

To start a CPI-C trace, enter the following from a command line:

sna -trace -c on

When your program finishes running, enter the following from the command line:

sna -trace -c off

sna -setlogs -t

This will take all the output generated from your trace and flush it into the **/var/sna/snaservice.X** file.

To format the data from that storage area into a readable file, enter this command:

trcrpt -d 390 /var/sna/snaservice.X > filename

filename is the name of an ordinary AIX file which will be created (or replaced) from the above command. If you skip the **> filename** part of the above command, output will go directly to the screen.

To determine what X should be, you need to perform the ls -l command on the **/var/sna** directory. Choose the largest file that was updated when you issued the setlogs -t command.

Using 27a instead of 390 will give you a trace of SNA events. Using 27b will give you information about SNA errors.

Reading the traces is easy. Each CPI Communications command generates two sections (at entry and exit) in the trace file. Some input and output parameters are shown in the trace, as well as return codes, error numbers, and conversation states.

Analyzing CPI Communications Return Codes: The return codes are used to report whether or not the program could execute the command, and what went wrong if execution failed. Some return codes will cause your program to enter reset state (essentially self-destruct) or enter receive state (usually because the program received a send error indication). Not all return codes result from errors in a conversation.

Note: See Appendix B, "Return Codes and Secondary Information" on page 661 for a list of return codes.

Analyzing AIX Errnos: *errnos* are SNA Server/6000's way of giving a user of an API information about what went wrong with a command. *errno* is a global variable which is defined in the file **/usr/include/errno.h**. To use this variable, include this file in the program.

The CPI Communications trace facility will also provide any *errno* which happened to occur as a result of a call.

For the most part, CPI Communications return codes are enough to deduce what went wrong with a particular call. But when CM_PRODUCT_SPECIFIC_ERROR is returned, the *errno* will provide additional information.

These are some of the more common *errnos*, which are not matched by CPI Communications return codes. The *errno* number itself is in parentheses.

- EINTR (4): While processing a call, the process the program was running under was interrupted. You will likely never get this error code unless you have set up an interrupt handler to avoid killing the TP in the event of an interruption.

  Note: CPI Communications will re-issue the command upon interruption, so this is a rare error. If the command is repeatedly interrupted, CPI Communications returns this error. One solution is to set up the interrupt handler to allow CPI Communications to stay running once interrupted.

- EBADF (9): This is a bad file descriptor error, which sometimes occurs when a call is made after the remote program has deallocated abnormally without issuing a Deallocate call. The local program has not issued a call, which would pick up a CM_RESOURCE_FAILURE return code. Check the remote program to ensure the program logic does not cause the program to end before doing all the processing required on both sides.
- EACCESS (13): This is the "permission denied" error. It usually indicates that the side information profile contains a parameter that does not allow SNA to put together a good attach. Maybe the link station referenced by the profile is not active.

Other *errnos* are more rare, and are easier to debug. *errnos* numbered 101 and higher can be found in **/usr/include/luxsna.h**. This file contains SNA Server/6000-specific error numbers (most of which CPI Communications translates to return codes). Lower-numbered *errnos* are related to AIX problems, and are defined in the file **/usr/include/sys/errno.h**.

Sometimes a program will just hang. This is probably a confirmation problem. A Confirm call, whether it stems from a Confirm, Send, Prepare_To_Receive, or Deallocate call, will wait until it gets a Confirmed or Send_Error call in response. A Deallocate call with *deallocate_type* of CM_DEALLOCATE_ABEND is also a possible response, though that stops a program. Ensure that all Receive calls on the

remote side look for confirmation requests. If a program is hanging when it should be allocating, it may have trouble getting a connection going. Starting the session manually will help on some systems. It is also possible that the side information profile contains an error.

When Allocation Requests Are Sent

Because SNA Server/6000 buffers data transmitted to the remote LU, the allocation request generated by Allocate is not sent to the remote node until the local LU's send buffer is flushed as the result of Send_Data, Flush, Receive, Prepare_To_Receive, Send_Error, Deallocate, Confirm, or other calls.

Allocation does not take place until the buffer has been flushed. Therefore, the remote program does not start until the allocation information reaches the remote machine. Also, an error in the allocation request (like an incorrect transaction program name) might not show up until a return code is received from a Receive call or another call that flushes the buffer.

Deviations from the CPI Communications Architecture

SNA Server/6000 supports CPI Communications calls with these distinctions:

- CPI Communications calls on SNA Server/6000 are supported by the C language only.
- On releases prior to Version 3 Release 1 of SNA Server/6000 do not log or transmit data associated with the Set_Log_Data call.
- SNA Server/6000 does not support conversations with a *sync_level* characteristic of CM_SYNC_POINT.

  Note: CPI-C 2.0 conformance.

Security Using CPI Communications and AIX

Security is one of the more difficult concepts when using AIX. It does, however, boil down to a basic idea:

> The source program is responsible for selecting a level of security to include with the allocation request. This request can include a user ID and/or password. The target machine (not the program) is responsible for maintaining two separate security checks of that incoming request. First, it checks to make sure the requesting program is allowed to use the session. And second, it checks that the requesting program is allowed to start the specified target program.

Okay, so it is not all that basic. In AIX SNA Server/6000, security acceptance is all done through profiles, and security requests are made through source transaction programs. Below is an algorithm for determining your security setup.

Note: Setting all security levels to none makes everything very easy if you have a secure system. However, since many systems allow access through dial-in modems or off-site links, protection of certain transaction programs is a good way to protect important information.

- Source-Side Security: The key value is the conversation security type, which defaults to XC_SECURITY_SAME in CPI-C, and can be set through the XCSCST command (after executing CMINIT and before executing CMALLC).

Security Type:

None: No security information is provided with the allocation request.

Same: A user ID and the already-verified notification are provided with the allocation request. (This user ID will default to the user ID of the process the source program is running under, unless this is a target program started using the “same” option, in which case, it will be the user ID that started the original source program. Note that this would be used by a target program only if it were starting a new source conversation.

Program: A user ID and password are provided with the allocation request.

- Target-Side Security:

  Key values include the security levels of the LU 6.2 Partner LU Profile, which has a partner LU name that matches the LU on the remote machine attempting to allocate a conversation and the Transaction Program Profile related to the target program referenced by the remote TP name set by the source program on the allocation request. Both these profiles act in tandem to create conversation-level security.

  Partner LU Profile Conversation security level field:

  None: Any allocation request will be accepted.

  Note: A request cannot pass this test with security information present. So the source LU, rather than needlessly sending this information, downgrades this security level to none and strips off the user ID and password, if present. This automatic downgrade will show up as an informational message in an API trace file, if you choose to run an API trace. The user will not have to alter the request as it is re-sent automatically with no security information present.

  Conversation: Allocation requests will be accepted unless the user ID and/or password are invalid. If the incoming request is type **program**, and a conversation security access list is defined in the Local LU profile representing the LU that is the target for the allocation request, the user ID is checked against that list (and the request is rejected if the ID is not present). The user ID is checked against valid user IDs on that machine (in the file /etc/security/passwd), regardless of whether a security access list is provided. The password is checked against the encoded passwords in the file /etc/security/passwd. The request is rejected if the password does not match the machine's password for that user ID. If the incoming request is type **same**, security information is stripped off and the request is re-sent without security parameters in exactly the same manner as it would be if this field were set to **none**.

  Already Verified: Allocation requests with an already-verified indicator will be accepted unless they do not provide a user ID. Allocation requests with a user ID and a password will have user IDs checked against the Conversation Security Access List, if present, and the list of user IDs in the file **/etc/security/password**. The password is checked against the encoded passwords in the file **/etc/security/password**. The request is rejected if the password does not match the machine's password for that user ID.

Transaction Program Profile security level field (only requests that have passed the LU session-level test are checked using this profile):

None: All requests are accepted.

Conversation: All requests with an incoming security level of **same** or **program** are accepted unless the user ID does not appear in the file **/etc/security/password**.

Note: If security information was stripped off due to a downgrade caused by the value in the partner LU profile's conversation security level field, the request will fail.

Access: All requests with an incoming security level of **same** or **program** are accepted, unless the user ID does not appear in the file **/etc/security/password**. If there is a resource security access list defined in this transaction program profile, the user ID is checked against that list. If the user ID is not on that list, the request will fail.

Note: If security information was stripped off due to a downgrade caused by the value in the partner LU profile's security level field, the request will fail.

Compilation

On AIX, compilation of CPI Communications programs requires three files:

- The appropriate library, either **libcpic.a** or **libcpic_r.a** must be in the directory **/usr/lib**. The library **libcpic_r.a** is a thread-safe library and is only present in SNA Server/6000 Version 3 Release 1 and higher.

  This is the case if SNA Server/6000 is installed properly.
- The include file **cmc.h** must be in the directory **/usr/include**. This is the case if SNA Server/6000 is installed properly.
- A program must have the statement **#include <cmc.h>** before making any CPI Communications calls. It must reference all CPI Communications commands properly. All command names must be in lowercase and contain the proper number and type of arguments. To use macros associated with product-specific errors, include the statement **#include <luxsna.h>**.

To compile, execute the following command from the command line. This example assumes the program name is **program.c** and the executable will be named **prog**.

```
cc program.c -o prog -lcpic
```

To compile a multi-threaded program a command such as the following should be used.

```
cc_r program.c -o prog -lcpic_r
```

Running a Transaction Program

Follow these steps to start a transaction program:

1. Before a transaction program can be run, SNA server must be started on both machines. To do this, type the following from the command line:

   ```
   sna -s sna
   ```

2. Bring up a link between the two machines. To do this, both machines must have profile sets. One machine must have a calling station and the other a listening station. Note that it does not matter which machine calls and which machine listens. Source programs can be run from both. Type the following from the command line:

   sna -s l -p name_of_calling_link_station

   Wait for the CP-CP session to become active on both machines. CP-CP sessions will only come active if at least one machine is an NN. To check the status of SNA sessions, type the following from the command line:

   sna -d s

   If active CP-CP sessions do not exist on both machines, there is a problem with the profile set.

AIX Extension Calls

SNA Server/6000 extends CPI Communications with five additional calls that enable a program to obtain or change the values of the conversation-security characteristics.

Note: If these calls are used, the program must be modified before it can be run on a different CPI Communications platform that does not provide the same extension calls with the same syntax. Where a similar extension call exists in other CPI Communications environments, the same call name and syntax are used in AIX to aid portability.

Three of these calls are used to set the conversation security characteristics used with the Allocate (CMALLC) call. The other two calls are used obtain the current value of these characteristics, except for *security_password*. The password can be set, but, to reduce the risk of unintentional or unauthorized access to passwords, it cannot be extracted.

Table 19 lists the AIX extension calls and briefly describes their functions.

Table 19. List of SNA Server/6000 Extension Calls for CPI Communications

Call and Pseudonym	Description
XCECST Extract_Conversation_Security_Type	Returns the current value of the *conversation_security_type* characteristic.
XCECSU Extract_Conversation_Security_User_ID	Returns the current value of the *security_user_ID* characteristic.
XCSCSP Set_Conversation_Security_Password	Sets the value of the *security_password* characteristic.
XCSCST Set_Conversation_Security_Type	Sets the value of the *conversation_security_type* characteristic.
XCSCSU Set_Conversation_Security_User_ID	Sets the value of the *security_user_ID* characteristic.
XCSSB Set_Signal_Behavior	Determines the behavior of CPI Communications calls if they are interupted by a signal.

Note: As with all CPI Communications calls in AIX SNA Server/6000, these calls are available only in the C language. The function names representing these calls are the same as the call names shown, but in lowercase characters.

Extract_Conversation_Security_Type (XCECST)

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

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

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

- XC_SECURITY_NONE
  No access security information is associated with the allocation request.
- XC_SECURITY_SAME
  If the local TP was started as a target TP and is now acting as source TP for another conversation, the user ID (if any) on the inbound allocation request that started the local TP will be included on the allocation request sent to the partner LU, along with an "already verified" indication.

  If the local TP was not started as a target TP, the user ID under which it is executing will be included on the allocation request sent to the partner LU, along with an "already verified" indication.
- XC_SECURITY_PROGRAM
  The access security information included with the allocation request sent to the partner LU will consist of the *conversation_security_user_ID* and *security_password* characteristics. These characteristics must have been set using the XCSCSU and XCSCSP calls.

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

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

State Changes

This call does not cause a state change.

Usage Notes

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

Extract_Conversation_Security_User_ID (XCECSU)

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

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

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

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

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

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

State Changes

This call does not cause a state change.

Usage Notes

1. If a *return_code* other than CM_OK is returned on this call, the values contained in the *security_user_ID* and *security_user_ID_length* variables are not meaningful.
2. This call does not change the security user ID for the specified conversation.
3. The value of the security_user_ID is set to the user_ID in the last incoming attach request, if that request contained a user ID. This is the case following a CMACCP or a CMINIT call. If no user ID was put by the attach, the characteristic is set to the user ID of the process running the transaction program. This processing is only performed after the CMINIT call.
4. Programs running under SNA Server/6000 Version 3 Release 1 can instead call CMESUI.

Set_Conversation_Security_Password (XCSCSP)

A program issues the Set_Conversation_Security_Password (XCSCSP) call to set the access security password for a conversation. A password is necessary to establish a conversation that uses a *conversation_security_type* of XC_SECURITY_PROGRAM.

Set_Conversation_Security_Password can be called only for a conversation in **Initialize** state having a *conversation_security_type* of XC_SECURITY_PROGRAM.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

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

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

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

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

State Changes

This call does not cause a state change.

Usage Notes

1. If a *return_code* other than CM_OK is returned on this call, the *security_password* characteristic remains unchanged.
2. The *conversation_security_password* is not initialized when the program calls Initialize_Conversation (CMINIT). This password can only be set by means of the XCSCSP call.
3. If the *conversation_security_type* characteristic is XC_SECURITY_PROGRAM when the program calls Allocate (CMALLC), SNA Server/6000 obtains the access security password for the allocation request from the *security_password* characteristic. If the *conversation_security_type* is other than XC_SECURITY_PROGRAM, SNA Server/6000 ignores the *security_password* characteristic when the program calls Allocate.
4. If an invalid security password is specified, it is not detected on this call; it is detected by the partner LU when it receives the allocation request. The partner LU returns an error indication to the local LU, which reports the error to the program by means of the CM_SECURITY_NOT_VALID return code on a subsequent call to CPI Communications.
5. Specify *security_password* using the native encoding for AIX, which is 8-bit ASCII. SNA Server/6000 converts the password to EBCDIC before including it on allocation requests sent to partner LUs.
6. Programs running under SNA Server/6000 Version 3 Release 1 can specify a value between 1 and 10 for security_password_length.
7. Programs running under SNA Server/6000 Version 3 Release 1 can instead call CMSCSP.

Set_Conversation_Security_Type (XCSCST)

A program issues the Set_Conversation_Security_Type (XCSCST) call to set the security type for the conversation. This call can be used to override the security type of XC_SECURITY_SAME that is assigned when the conversation is initialized.

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

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

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

- XC_SECURITY_NONE
  No access security information is associated with the allocation request.
- XC_SECURITY_SAME
  If the local TP was started as a target TP and is now acting as source TP for another conversation, the user ID (if any) on the inbound allocation request that started the local TP is included on the allocation request sent to the partner LU, along with an “already verified” indication.

  If the local TP was not started as a target TP, the user ID, under which it is executing, is included on the allocation request sent to the partner LU, along with an “already verified” indication.
- XC_SECURITY_PROGRAM
  The access security information included with the allocation request sent to the partner LU will consist of the *conversation_security_user_ID* and *security_password* characteristics. These characteristics must have been set using the XCSCSU and XCSCSP calls.

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

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

State Changes

This call does not cause a state change.

Usage Notes

1. If a *return_code* other than CM_OK is returned on this call, the *conversation_security_type* characteristic remains unchanged.
2. When the program calls Initialize_Conversation (CMINIT), SNA Server/6000 initializes the *conversation_security_type* to XC_SECURITY_SAME. A program does not need to issue the Set_Conversation_Security_Type call unless the security type is to be other than XC_SECURITY_SAME.
3. If the program sets or defaults the *conversation_security_type* to XC_SECURITY_NONE or XC_SECURITY_SAME, any conversation security ID or password values set by XCSCSU or XCSCSP will be ignored by CMALLC. When the *conversation_security_type* characteristic is set to XC_SECURITY_SAME and the user is part of the system group, then the allocation request uses the *conversation_security_user_ID* set through this call.
4. If the program uses XCSCST to set *conversation_security_type* to XC_SECURITY_PROGRAM, it must then call Set_Conversation_Security_Password (XCSCSP) to set the *security_password* and Set_Conversation_Security_User_ID (XCSCSU) to set the *security_user_ID*. Otherwise, when the program calls Allocate, the initial values (a single space character) will be used for these characteristics.
5. If the program uses XCSCST to set the user ID to xc_security_same, the user ID stored by CPI-C will change to the user ID of the process running the transaction program.
6. Programs running under SNA Server/6000 Version 3 Release 1 can instead call CMSCST.

Set_Conversation_Security_User_ID (XCSCSU)

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

Set_Conversation_Security_User_ID can be called only for a conversation in **Initialize** state having a *conversation_security_type* of XC_SECURITY_PROGRAM or XC_SECURITY_SAME.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation ID.

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

security_user_ID_length *(input)*
Specifies the length of the security user ID, which must be 1–8 characters.

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

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

State Changes

This call does not cause a state change.

Usage Notes

1. If a *return_code* other than CM_OK is returned on this call, the *security_user_ID* characteristic remains unchanged.
2. When the program calls Allocate (CMALLC) and the *conversation_security_type* characteristic is XC_SECURITY_PROGRAM, SNA Server/6000 obtains the access security user ID for the allocation request from the *conversation_security_user_ID* characteristic. If the *conversation_security_type* is other than XC_SECURITY_PROGRAM, SNA Server/6000 ignores the *conversation_security_user_ID* characteristic, unless the user is part of the system group. When the program calls Allocate (CMALLC) and the *conversation_security_type* characteristic is set to XC_SECURITY_SAME and the user is part of the system group, then the allocation request uses the *conversation_security_user_ID* set through this call.
3. Specification of a security user ID that is invalid is not detected on this call. It is detected by the partner LU when it receives the allocation request. The partner LU returns an error indication to the local LU, which reports the error to the program by means of the CM_SECURITY_NOT_VALID return code on a subsequent CPI Communications call.
4. Specify *security_user_ID* using the native encoding for AIX, which is 8-bit ASCII. SNA Server/6000 converts the user ID to EBCDIC before sending it, with an allocation request, to a partner LU.
5. Programs running under SNA Server/6000 Version 3 Release 1 can instead call CMSCSU.
6. Programs running under SNA Server/6000 Version 3 Release 1 can specify a value between 1 and 10 for security_user_ID_length.

Set_Signal_Behavior (XCSSB)

The Set_Signal_Behavior(xcssb) call specifis how CPI Communications should handle an interupted CPI Communications verb in the event that a signal interrupts the CPI Communications TP thread. If a CPI Communications TP does not uses this verb, CPI Communications will by defaults attempt to complete the interupted verb.

Format

```
CALL XCSSB(conversation_ID,
           Signal_behavior,
           return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation ID.

Signal_behavior *(input)*
must be one of the following values.

- XC_SIGNAL_BEHVIOR_NO_RETRY
  if Signal_behvior has this value, CPI Communications will not attempt to complete the verb. If this value is specified, and interupted CPI Communications verb will return the error CM_PRODUCT_SPECIFIC_ERROR, and errno will be set to EINTR. In that case, the CPI Communications TP should use CMCANC to cancel the conversation, which will be in an inderterminate state.

- XC_SIGNAL_BEHAVIOR_INFINITE_RETRY

  If Signal_behavior has this value, CPI Communications will attempt to complete the verb, even if more signals are delivered to the CPI Communications TP thread. If this value is specified, control of execution will not be returned to the CPI Communications TP in the event of a signal.

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

- CM_OK
  Successful completion.
- CM_PROGRAM_PARAMETER_CHECK
  This value indicates one of the following:
  – The *conversation_ID* specifies an unassigned conversation ID.
  – The *Signal_Behavior* specifies a value less than XC_SGINAL_BEHAVIOR_NO_RETRY or XC_SIGNAL_BEHAVIOR_INFINITE_RETRY

State Changes

This call does not cause a state change.

Usage Notes

This call should only be made in SNA Server/6000 Version 3 Release 1.

Chapter 6. CPI Communications on CICS/ESA

This chapter summarizes product-specific information that the CICS programmer needs when writing application programs that contain CPI Communications calls.

CICS/ESA has provided support for CPI Communications since Version 3 Release 2.1. The *CICS/ESA Intercommunication Guide* deals with CICS intercommunication generally, and gives details of how to define links for APPC support.

This chapter is organized as follows:

- CICS/ESA Publications
- CICS/ESA Operating Environment
  - Conformance Classes Supported
  - Languages Supported
  - Pseudonym Files
  - Defining Side Information
  - How Dangling Conversations Are Deallocated
  - Scope of the Conversation_ID
  - Identifying Product-Specific Errors
  - Diagnosing Errors
  - When Allocation Requests Are Sent
  - Deviations from the CPI Communications Architecture
- CICS/ESA Extension Calls
- CICS/ESA Special Notes

CICS/ESA Publications

The following CICS/ESA publications contain detailed product information:

- *CICS/ESA Intercommunication Guide*, SC33-1181
- *CICS/ESA Application Programming Guide*, SC33-1169
- *CICS/ESA Application Programmer's Reference*, SC33-1170
- *CICS Library Guide*, GC33-1226

CICS/ESA Operating Environment

The CPI Communications Interface in CICS provides an alternative application programming interface (API) to existing CICS APPC communications support.

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

Alternatively, users might prefer to use CPI Communications in APPC networks that include multiple system platforms, where a common API's consistency is seen to be beneficial.

© Copyright IBM Corp. 1996, 1998

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

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

The following sections explain some special considerations that should be understood when writing applications for a CICS environment.

Conformance Classes Supported

CICS supports the following conformance classes:

- Conversations

  All conversations, with the exception of Extract_Maximum_Buffer_Size (CMEMBS)

- LU 6.2
- Recoverable Transactions

Refer to "Functional Conformance Class Descriptions" on page 746 for a complete description of functional conformance classes.

Languages Supported

The following SAA languages can be used on a CICS/ESA system to issue CPI Communications calls:

- C
- COBOL
- PL/I

System/370 Assembler can also be used even though it is not an SAA language.

Note: CICS/ESA requires that applications written for use with CPI Communications should be fully re-entrant. Exceptions to this are applications written in System/370 assembler; these must be quasi-reentrant. For guidance on quasi-reentrancy and multithreading, see the *CICS/ESA Application Programming Guide*.

The programming interfaces for COBOL, PL/I, and C are as described in "Programming Language Considerations" on page 111. All programs must set up a parameter list and obey the standard IBM linking convention.

Pseudonym Files

CICS provides a pseudonym file for each language supported.

Table 20. CICS Pseudonym Files for Supported Languages		
Language	**File name**	**Location**
COBOL	CMCOBOL	CICSxxx.SDFHCOB
PL/I	CMPLI	CICSxxx.SDFHPLI
C	CMC	CICSxxx.SDFHC370
Assembler	CMHASM	CICSxxx.SDFHMAC
Note: xxx is the CICS release number. (For example, 321 would mean Version 3 Release 2 Modification 1.)		

Defining Side Information

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

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

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

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

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

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

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

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

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

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

TPNAME(name)|XTPNAME(value)
The *TP_name* (transaction identifier in CICS) can be defined using either the TPNAME or XTPNAME parameter. Use TPNAME when the characters shown in Table 21 on page 411 can be used. Use XTPNAME to specify the hexadecimal values for characters that CICS does not allow for the TPNAME parameter.

When the conversation is initiated by Allocate (CMALLC), the *TP_name* (which will be either TPNAME or XTPNAME) is sent to the remote LU.

When a CICS/ESA system receives a request to attach a transaction, it searches it's transaction definitions to determine which transaction to attach. Those transaction definitions are defined by the CEDA DEFINE TRANSACTION(*name*) command, and may have either a TPNAME or a XTPNAME in a similar manner to partner transactions defined in the side information table. CICS/ESA will use the first four characters of *TP_name* to identify the transaction, but optionally you can use the global user exit XZCATT to convert a 64 character *TP_name* to four-characters.

Table 21 on page 411 gives details of character sets used in CICS to define the PARTNER operands. Where CICS names have to match with the variables listed in Table 61 on page 650, it is recommended that the character sets for CPI Communications, as defined in Table 21 on page 411, be used if portability is to be maintained for CICS.

<table>
<caption>Table 21. Defaults and Allowed Values for the CICS PARTNER Resource</caption>
<tr><th>Operand</th><th>Default</th><th>Mandatory</th><th>Type</th><th>Length</th><th>Range of values</th></tr>
<tr><td>PARTNER</td><td></td><td>yes</td><td>chars</td><td>8</td><td>A– 0–9</td></tr>
<tr><td>GROUP</td><td>Current group</td><td>no</td><td>chars</td><td>1-8</td><td>A– 0–9 @ # $
Lowercase changed to uppercase</td></tr>
<tr><td>NETWORK</td><td>Undefined</td><td>no</td><td>chars</td><td>1-8</td><td>A– 0–9 @ # $
Lowercase changed to uppercase</td></tr>
<tr><td>NETNAME</td><td>Undefined</td><td>yes</td><td>chars</td><td>1-8</td><td>A– 0–9 @ # $
Lowercase changed to uppercase</td></tr>
<tr><td>PROFILE</td><td>DFHCICSA [1]</td><td>no</td><td>chars</td><td>1-8</td><td>A– 0–9 ¢ @ # . / - _ % & $? ! : | " = ¬ , ; < ></td></tr>
<tr><td>TPNAME[2]</td><td>Undefined</td><td>yes (or XTPNAME)</td><td>chars</td><td>1-64</td><td>A– 0–9 ¢ @ # . / - _ % & $? ! : | " = ¬ , ; < ></td></tr>
<tr><td>XTPNAME</td><td>Undefined</td><td>yes (or TPNAME)</td><td>hex chars</td><td>2-128</td><td>0-9 A-F excluding the hex byte X'40'</td></tr>
</table>

Note:

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

How Dangling Conversations Are Deallocated

If a CICS transaction terminates during a conversation, CICS tries to end the dangling conversation normally. If this fails, abnormal termination is performed.

For conversations with the *sync_level* set to CM_NONE or CM_CONFIRM, CICS attempts a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_FLUSH. If this call results in a state check, then the conversation is terminated with a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND.

For conversations with *sync_level* set to CM_SYNC_POINT, CICS attempts the equivalent of a Deallocate call with the *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL followed by an SAA resource recovery Commit call. If this fails because of a state check, the task abends, and the conversation is terminated. The equivalent of a Deallocate call with *deallocate_type* is set to CM_DEALLOCATE_ABEND.

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

Scope of the Conversation_ID

The scope of a *conversation_ID* within CICS/ESA is one CICS task. A *conversation_ID* is created when a task initializes or accepts a conversation. Thereafter, any CICS/ESA application running under this task can use *conversation_ID* to issue verbs against the conversation during its lifetime.

Identifying Product-Specific Errors

The CM_PRODUCT_SPECIFIC_ERROR return code results in one of the following informational error messages:

- DFHCP0742—The session is not available for CPI Communications as it is already in use by another process.
- DFHCP0743—CPI Communications cannot be used, as the transaction was initiated by ATI.
- DFHCP0750—An unrecognized profile name was supplied in partner resource *sym_dest_name*.

These messages are sent to the CCPI transient data queue.

There are no state transitions connected with CM_PRODUCT_SPECIFIC_ERROR.

Diagnosing Errors

On detecting error conditions that result in a return code of CM_PRODUCT_SPECIFIC_ERROR, CM_PARAMETER_ERROR, CM_PROGRAM_PARAMETER_CHECK, or CM_PROGRAM_STATE_CHECK, CICS sends an explanatory message to the CCPI transient data queue, which is used for logging CPI Communications messages.

If CICS receives an unrecognized sense code from the partner program, CICS returns one of these return codes:

CM_DEALLOCATED_ABEND
CM_DEALLOCATED_ABEND_BO

or one of these:

CM_PROGRAM_ERROR_PURGING
CM_SVC_ERROR_PURGING

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

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

When Allocation Requests are Sent

The allocation request is not sent as part of the Allocate call. It is buffered to be sent later, when a subsequent call causes the buffer to be flushed.

Deviations from the CPI Communications Architecture

On CICS/ESA systems, when the program is using CPI Communications to communicate with releases of CICS earlier than CICS/ESA Version 3.2, the program's conversation state after a backout is one of the following:

- If the program initiated the backout, its side of the conversation is placed in **Send** status.
- If the program did not initiate the backout, its side of the conversation is placed in **Receive** status.
- If the program's side of the conversation was in **Defer-Deallocate** status when the backout occurred, the conversation is placed in **Reset** state.

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

CPI Communications applications in CICS cannot be SNA service programs and therefore cannot allocate on the mode names SNASVCMG or CPSVCMG. If they attempt to do this they will get CM_PARAMETER_ERROR.

CICS attempts to end dangling conversations normally, but if this fails then abnormal termination is performed.

CICS/ESA Extension Calls

CICS/ESA provides no CPI Communications extension calls.

CICS/ESA Special Notes

CICS programmers should note the following points when writing programs that issue CPI Communications calls:

- A CICS transaction started by automatic transaction initiation (ATI) cannot use CPI Communications calls on its principal facility.
- If a CICS transaction issues an Allocate call (CMALLC) and the status of the connection specified in the partner definition is INSERVICE RELEASED, CICS does not try to ACQUIRE the connection; a return code of CM_ALLOCATE_FAILURE_RETRY is returned to the application.
- CICS indicates USER_ID_IS_ALREADY_VERIFIED in the FMH5 header for outgoing attach requests. The USER_ID is also in the same header.
- The security requirements of incoming attach requests can be specified by CICS resource definition.
- The *TP_name* may contain the CICS four-character transaction identifier of the partner program if the partner LU is CICS.
- APPC transaction routing is supported for CPI Communications between a pair of CICS/ESA systems.

Chapter 7. CPI Communications on IMS/ESA

Programs running under IMS/ESA can use all the CPI Communications calls, extensions, and features provided by APPC/MVS. No special setup or restrictions apply. For a description of how to use CPI Communications under IMS/ESA, see Chapter 8, "CPI Communications on MVS/ESA" on page 417 and the following IMS/ESA Version 5 books:

- *IMS/ESA Application Programming: Transaction Manager*, SC26-8017-00
- *IMS/ESA Administration Guide: Transaction Manager*, SC26-8014-00
- *IMS/ESA Application Programming: Database Manager*, SC26-8015-01
- *IMS/ESA Application Programming: Design Guide*, SC26-8016-00

Additional product information can be found in the following IMS/ESA books:

- *IMS/ESA General Information*, GC26-3467-00
- *IMS/ESA Release Planning Guide*, GC26-8031-00
- *IMS/ESA Version 5 Licensed Program Specifications*, GC26-8040-00
- *IMS/ESA Customization Guide*, SC26-8020-00
- *IMS/ESA Utilities Reference: Transaction Manager*, SC26-8022-00
- *IMS/ESA Installation Volume 1: Installation and Verification*, SC26-8023-00
- *IMS/ESA Installation Volume 2: System Definition and Tailoring*, SC26-8024-00
- *IMS/ESA Administration Guide: System*, SC26-8013-00
- *IMS/ESA Operations Guide*, SC26-8029-00
- *IMS/ESA Operator's Reference*, SC26-8030-00
- *IMS/ESA Sample Operating Procedures*, SC26-8032-00
- *IMS/ESA Messages and Codes*, SC26-8028-00
- *IMS/ESA Failure Analysis Structure Tables (FAST) for Dump Analysis*, LY27-9621-00
- *IMS/ESA Diagnosis Guide and Reference*, LY27-9620-00
- *IMS/ESA LU 6.1 Adapter for LU 6.2 Applications: Program Description/Operations*, SC26-3061-01
- *IMS/ESA Master Index and Glossary*, SC26-8027-00
- *IMS/ESA Summary of Operator Commands*, SC26-8042-00

All IMS/ESA Version 5 books are available on CD-ROM; you can order IBM Online Library: Transaction Processing and Data (SK2T-0730) or MVS Collection (SK2T-0710).

For IMS/ESA Version 5, there is also *IMS/ESA Softcopy Master Index*, which is a master index of the IMS library. It is available only in BookManager format on either of the CD-ROMs listed above.

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

© Copyright IBM Corp. 1996, 1998

Chapter 8. CPI Communications on MVS/ESA

This chapter summarizes the product-specific information that the MVS programmer needs when writing application programs that contain CPI Communications calls.

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

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

This chapter is organized as follows:

- MVS/ESA Publications
- MVS/ESA Operating Environment
  - Conformance Classes Supported
  - Languages Supported
  - Pseudonym Files
  - Defining Side Information
  - How Dangling Conversations Are Deallocated
  - Scope of the Conversation_ID
  - Identifying Product-Specific Errors
  - Diagnosing Errors
  - When Allocation Requests Are Sent
  - Deviations from the CPI Communications Architecture
- MVS/ESA Extension Calls
- MVS/ESA Special Notes

MVS/ESA Publications

- *MVS/ESA Programming: Writing Transaction Programs for APPC/MVS*, GC28-1471
- *MVS/ESA Planning: APPC Management*, GC28-1503
- *MVS/ESA Programming: Writing Servers for APPC/MVS*, GC28-1472
- *MVS/ESA Programming: Writing Transaction Schedulers for APPC/MVS*, GC28-1465
- *MVS/ESA APPC/MVS Handbook for the OS/2 System Administrator*, GC28-1504

© Copyright IBM Corp. 1996, 1998

MVS/ESA Operating Environment

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

The following requirements apply to TPs that are written to run in problem-program state:

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

The general requirements shown in Table 22 apply to CPI Communications calls invoked by any TP. They include requirements (such as locks not allowed) that are only of concern to TPs running in supervisor state or with PSW key 0-7.

Table 22. General Requirements for CPI Communications Calls on MVS

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

Programs using CPI Communications on MVS, other than those written in REXX, must use one of the following methods to access the APPC/MVS system services:

- Link-edit the program with the load module ATBPBI, which is provided in SYS1.CSSLIB.
- Issue the MVS LOAD macro for the APPC/MVS service to obtain its entry point address. Use that address to call the APPC/MVS service.

TSO/E release 2.3 provides this service for interpreted REXX programs. TSO/E releases 2.3.1 and above provide this service for compiled REXX programs.

TPs that call CPI Communications services while in task mode should not have any enabled unlocked task (EUT) functional recovery routines (FRRs) established.

The following sections discuss special considerations that should be understood when writing applications for an MVS environment.

Conformance Classes Supported

MVS supports the following conformance classes:

- Conversations

  All conversations, with the exception of Extract_Maximum_Buffer_Size (CMEMBS)

- LU 6.2

Refer to "Functional Conformance Class Descriptions" on page 746 for a complete description of functional conformance classes.

Languages Supported

The following languages can be used to issue CPI Communications calls:

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

In addition, any high-level language that conforms to the following linkage conventions may be used to issue CPI Communications calls on MVS:

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

On return from the service, general and access registers 2–14 are restored. Registers 0, 1 and 15 are not restored.

Pseudonym Files

Pseudonym files (also called interface definition files or IDFs) for the CPI Communications calls are provided, as described in Table 23. Pseudonym files define calls, parameters, and variable values in the SAA languages.

As shown in Table 23, the CPI Communications pseudonym files in SYS1.SAMPLIB are not named according to the CPI-C file names. IBM recommends that a pseudonym file be renamed to the CPI-C name when being placed in a high-level language macro library.

Table 23. CPI Communications Pseudonym Files on MVS

Language	In SYS1.SAMPLIB Member	CPI-C File Name
C	ATBCMC	CMC
COBOL	ATBCMCOB	CMCOBOL
FORTRAN	ATBCMFOR	CMFORTRN
PL/I	ATBCMPLI	CMPLI
REXX	ATBCMREX	CMREXX
RPG	ATBCMRPG	CMRPG

CSP pseudonym file CMCSP COPY is shipped by Cross System* Product, Release 3.3.0 and above.

Defining Side Information

In MVS/ESA, CPI Communications programs must provide a symbolic destination name on an Initialize_Conversation call. For each *sym_dest_name* used on a CMINIT call, provide an entry containing the following in a side information file on MVS:

- TP name
- Logon mode name
- Partner LU name

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

How Dangling Conversations Are Deallocated

Programs should deallocate conversations before ending, and should have recovery routines to deallocate conversations in the event of abnormal termination. If a program ends without deallocating its conversations, partner programs might continue to send data or wait to receive data. MVS deallocates any dangling conversation. If this is a mapped conversation, the partner on the other end of the conversation receives a return code of CM_RESOURCE_FAILURE_NO_RETRY. If it is a basic conversation, the partner on the other end of the conversation receives a return code of either CM_RESOURCE_FAILURE_NO_RETRY or CM_DEALLOCATED_ABEND_SVC.

Scope of the Conversation_ID

MVS considers the scope of a transaction program to be the home address space. MVS allows programs to share a single conversation, represented by a *conversation_ID*, across multiple tasks or SRBs in the home address space.

The application must pass the *conversation_ID* between tasks and SRBs (MVS does not provide a service for this purpose). Only one task or SRB can have control of the conversation at a given time, and only one CPI Communications call can be outstanding from one address space for a given conversation at a time. (When a CPI Communications call is outstanding, a return code of CM_PRODUCT_SPECIFIC_ERROR will be given for CPI Communications calls issued from the same address space.)

The only exception is the Deallocate call with a *deallocate_type* of CM_DEALLOCATE_ABEND, which can be issued from one task when a CPI Communications call is outstanding from another task for the same conversation. In that case, the outstanding call receives a return code of CM_PRODUCT_SPECIFIC_ERROR.

Identifying Product-Specific Errors

CPI Communications defines a return code called CM_PRODUCT_SPECIFIC_ERROR for each call. MVS returns this code when either:

- An MVS-specific error occurred, or
- APPC/MVS is not active.

On MVS, product-specific errors cause state transitions in the following situations:

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

When a return code of PRODUCT_SPECIFIC_ERROR is returned to the local program in response to a CPI Communications call, the code often has an accompanying symptom record in the logrec data set. If APPC/MVS is not active, however, a symptom record is not produced. In this case, your TP can call the MVS-specific Error_Extract callable service, which returns decimal code 64 when APPC/MVS is not active. No further diagnostic information is available when APPC/MVS is not active.

When a symptom record is recorded in the logrec data set, section 3 of the symptom record contains the primary symptom string for the product-specific errors:

Table 24. Symptom String for Product-Specific Errors on MVS. (Section 3 of the Symptom Record in a Logrec Data Set).

Symptom	Description
PIDS/5752SCACB	Product identifier
RIDS/ATBxxxxx	CSECT name
RIDS/ATBxxxxx#L	Load module name
LVLS/ddd	Product level
PCSS/ATBxxxx or CMxxxx	The statement that caused the error
PRCS/dddddddd	The return code returned to the caller of the service
FLDS/REASON VALU/Hdddddddd	The unique reason code identifying the product-specific error

Section 5 of the symptom record contains the following information for the product-specific error:

- The job or user name, in EBCDIC, for the home address space of the caller
- An EBCDIC description of the error (up to 80 characters)

Look for symptom FLDS/REASON VALU/Hdddddddd in section 3 of the symptom record for the reason code identifying the error, which is one of those appearing in Table 25 on page 422.

Table 25 (Page 1 of 2). Reason Codes for Product-Specific Errors on MVS

Reason Code	Message Text	Explanation
00000001	APPC SERVICE REQUESTED WHILE SYSTEM LOCK HELD.	A user requested an APPC/MVS service while a system lock was held.
00000002	UNRECOGNIZED REQUEST.	The system request from the caller is not one of the APPC CPIC or APPC LU 6.2 calls. The program might be using an incorrect level of the stub routine.
00000003	APPC DATA STRUCTURES FOR THE TP ARE IN USE BY ANOTHER PROCESS.	APPC/MVS data structures for the TP are in use by another process.
00000004	SYSTEM CANNOT PROCESS A CALL. A CLEANUP_TP REQUEST IS IN PROGRESS.	A program called the Cleanup_TP service for the TP that owns the conversation. The conversation is no longer available to the TP.
00000005	APPC/MVS COULD NOT RETRIEVE SIDE INFORMATION.	An error occurred when APPC/MVS tried to retrieve side information from the side information file.
00000006	ERROR RETRIEVING SECURITY INFORMATION.	An error occurred when APPC/MVS tried to obtain information about the caller's security environment from RACF. The RACF return code and reason code appear in section 5 of the symptom record. See *RACF V1 R9.2 Messages and Codes, SC38-1014* for explanations of the return and reason codes from RACF.
00000008	ADDRESS SPACE CANNOT USE THE SYSTEM BASE LU.	A program tried to allocate or initialize a conversation. The program is running in an address space that is not connected to a scheduler. Therefore, APPC/MVS must use the system default base LU as the source LU for the conversation. The address space in which the program is running was prohibited from using the system default base LU.
00000009	NO BASE LU DEFINED FOR SCHEDULER.	A program tried to allocate or initialize a conversation. The program is running in an address space that is connected to a scheduler. Therefore, APPC/MVS uses the base LU that is defined for that scheduler in the APPCPM*xx* parmlib member. None of the LUs assigned to the scheduler were designated as the base LU.
0000000A	SCHEDULER EXTRACT EXIT COULD NOT IDENTIFY ACTIVE TP.	An APPC/MVS service was invoked in an address space that has more than one active TP. APPC/MVS invoked the transaction scheduler extract exit, which must return the TPID of the active transaction program. This exit ended abnormally or returned with a nonzero return code.
0000000B	STORAGE NOT AVAILABLE FOR APPC INTERNAL STRUCTURES.	APPC could not obtain enough storage to process the requested service.
0000000C - 0000000D	AN INTERNAL FAILURE OCCURRED IN APPC PROCESSING.	APPC data structures for the TP are in use by another process. The system does not request a dump.
0000000E	SERVICE INTERRUPTED BY CALL TO CLEANUP_TP SERVICE.	A service was interrupted because a Cleanup_TP was issued against the TP that owns the conversation.
0000000F	STORAGE NOT AVAILABLE FOR APPC INTERNAL STRUCTURES.	APPC could not obtain storage for internal structures. APPC writes a symptom record for all failures to obtain storage.

Table 25 (Page 2 of 2). Reason Codes for Product-Specific Errors on MVS		
Reason Code	**Message Text**	**Explanation**
00000010	AN INTERNAL FAILURE OCCURRED IN APPC PROCESSING.	An internal failure occurred in APPC processing.
00000011 - 00000013	AN INTERNAL FAILURE OCCURRED IN APPC PROCESSING.	An internal error occurred in APPC/MVS processing.
00000014	INFORMATION ABOUT LOCAL LU WAS NOT AVAILABLE TO APPC/MVS.	APPC/MVS could not locate the local LU.
00000015 - 00000016	AN INTERNAL FAILURE OCCURRED IN APPC PROCESSING.	An internal error occurred in APPC/MVS processing.
00000017	PROCESSING FOR SERVICE INTERRUPTED BY DEALLOCATE_ABEND.	The requested callable service was interrupted because a Deallocate (ABEND) was issued against the conversation.
00000018	AN INTERNAL FAILURE OCCURRED IN APPC PROCESSING.	An internal error occurred in APPC/MVS processing.
00000019	A PREVIOUS ERROR LEFT THE CONVERSATION IN AN UNDEFINED STATE.	A previous error left the conversation in an undefined state. Issue a Deallocate with a deallocate_type of ABEND on the conversation to free up the resources for the conversation. This action will cause APPC/MVS to end the current session.
0000001A	AN UNEXPECTED RESOURCE FAILURE OCCURRED.	APPC/MVS found a resource failure when processing a service that does not return the resource_failure_retry or resource_failure_no_retry return codes.
0000001B	SERVICE INTERRUPTED BY CALL TO CLEANUP_TP SERVICE.	The service was interrupted because a Cleanup_TP was issued against the TP that owns the conversation.
0000001C- 0000001D	AN INTERNAL FAILURE OCCURRED IN APPC PROCESSING.	An internal error occurred in APPC/MVS processing.
0000001E - 0000001F	PROCESSING FOR SERVICE INTERRUPTED BY DEALLOCATE_ABEND.	A service was interrupted because a Deallocate (ABEND) was issued against the conversation.
00000021	ADDRESS SPACE IN WHICH TP IS RUNNING CANNOT USE SYSTEM BASE LU.	A program tried to allocate or initialize a conversation. The program is running in an address space that is prohibited from using the system base LU that is defined in the APPCPM*xx* parmlib member.
00000022	NO SYSTEM BASE LU DEFINED TO APPC/MVS.	A program tried to allocate or initialize a conversation. The attempt was rejected because the program did not specify a local LU name, and no default system base LU exists.
00000023	AN INTERNAL FAILURE OCCURRED IN APPC PROCESSING.	An internal error occurred in APPC/MVS processing.
00000024	BUFFER STORAGE NOT AVAILABLE FOR RECEIVE PROCESSING.	APPC/MVS could not obtain buffer storage to receive data sent by a partner TP.

Diagnosing Errors

The TP message log can be used for recovery and problem determination when an error occurs while a TP is processing. The TP message log can be controlled through parameters from the TP profile and from the APPC/MVS transaction scheduler parmlib member. When these parameters are used to set up the TP message log, the log can be accessed after a TP stops running, or the ASBSCHWL write log routine can be invoked (after a job step in the TP profile JCL)

to access the message log between job steps. The write log routine allows viewing messages for the previous job step.

For information on how to specify parameters used in the TP message log definition and view the message log, see *MVS/ESA Planning: APPC Management*.

When Allocation Requests Are Sent

Allocation requests are buffered until enough data is accumulated or the sender flushes the data; then the request is sent.

Deviations from the CPI Communications Architecture

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

- MVS/ESA does not log data associated with outgoing and incoming Send_Error and Deallocate (*Deallocate_type*= CM_DEALLOCATE_ABEND) calls.
- MVS/ESA allows a blank partner LU name to be specified on calls to the CMSPLN service. If the Allocate (CMALLC) call is issued when the partner LU name is set to blanks, the system considers the partner LU name to be the name of the local LU from which CMALLC is called. MVS also allows a blank partner LU name to be specified as an entry in the side information.
- When a conversation crosses a VTAM network, the Receive call does not return data until an entire logical record arrives at the local system. For basic conversations, this behavior may cause the Receive call to return unexpected results.

  For example, a Receive call with receive_type set to CM_RECEIVE_IMMEDIATE returns a return code of CM_UNSUCCESSFUL if the entire logical record has not arrived at the local system, even if enough of the logical record has arrived to satisfy the Receive call's requested length. Similarly, a Receive call with receive_type set to CM_RECEIVE_AND_WAIT will wait until the remainder of the logical record is received by the local system, even if enough of the logical record has arrived to satisfy the requested length. CMRCV works properly in conversations between LUs controlled by APPC/MVS in the same MVS system image.
- For conversations crossing a VTAM network, the Test_Request_To_Send_Received (CMTRTS) call always sets Request_To_Send_Received to CM_REQ_TO_SEND_NOT_RECEIVED when Return_Code=CM_OK, regardless of whether the remote programs have sent such requests to the local programs. CMTRTS works properly in conversations between LUs controlled by APPC/MVS in the same MVS system image.
- MVS/ESA does not support conversations with a *sync_level* characteristic of CM_SYNC_POINT.
- The character set restrictions on LU names, VTAM mode names, and TP names differ slightly from those prescribed in Appendix A, "Variables and Characteristics" on page 641.

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

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

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

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

Other deviations from the CPI Communications architecture are highlighted in green ink throughout this publication.

MVS/ESA Extension Calls

Although MVS/ESA does not provide CPI Communications extension calls, MVS/ESA does provide additional callable services that perform similar and additional communications functions. See "APPC/MVS Services" on page 425 for more information on these MVS-specific callable services.

MVS/ESA Special Notes

The following sections contain information that MVS programmers should consider when writing programs that issue CPI Communications calls:

TP Profiles

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

MVS Performance Considerations

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

APPC/MVS Services

In addition to CPI Communications calls, MVS also provides APPC/MVS callable services that are specific to MVS and provide similar and additional communications functions. MVS-specific calls begin with the characters ATB (ATBxxxx).

Programs on MVS can issue both CPI Communications (CMxxxx) and APPC/MVS (ATBxxxx) calls from the same program (as long as both pseudonym files are included in the program). The *conversation_ID* returned from a CMINIT or ATBALLC call is available to all subsequent CMxxxx or ATBxxxx calls for the conversation. For all conversation characteristics set by CMINIT, ATBALLC provides an equivalent parameter or sets the same default value. Calls to

APPC/MVS services do not change any conversation characteristics previously established by a CPI Communications call.

The following is a summary of MVS-specific services and functions:

Functions similar to conversation-level non-blocking calls: Many APPC/MVS services have an option to allow asynchronous execution. For more information about those services, see *MVS/ESA Application Development: Writing Transaction Programs for APPC/MVS*.

Functions similar to server calls: Allocate queue services allow you to set up “servers” that can process multiple inbound allocate requests. For information about those services, see *MVS/ESA SP V5 Writing Servers for APPC/MVS*.

Functions similar to secondary information calls: The Error_Extract service allows you to return detailed information about errors indicated by return codes from CPI Communications calls and LU 6.2 services. For more information about Error_Extract, see *MVS/ESA Application Development: Writing Transaction Programs for APPC/MVS*.

Additional functions: LU 6.2 services provide functions that are similar to CPI Communications calls, with some functions that are specific to APPC/MVS. For example, the APPC/MVS Send_Data (ATBSEND) service lets a program send data from an MVS/ESA data space. For more information about the APPC/MVS services, see *MVS/ESA Application Development: Writing Transaction Programs for APPC/MVS*.

Chapter 9. CPI Communications on Networking Services for Windows

This chapter contains information about the IBM APPC Networking Services for Windows implementation of, and extensions to, CPI Communications. The topics covered are:

- Networking Services for Windows Publications
- Conformance Classes Supported
- Languages Supported
- Pseudonym Files Provided by Networking Services for Windows
- Linking with the CPI-C Library
- Memory Considerations
- Defining Side Information
- Usage Notes for Mode_Name and TP_Name
- Deallocating Dangling Conversations
- Diagnosing Errors
- Deviations from the CPI Communications Architecture

Networking Services for Windows Publications

The following publications contain detailed product information:

- *IBM APPC Networking Services for Windows: Getting Started*, SC31-8124
- *IBM APPC Networking Services for Windows: Administrator's Guide*, SC31-8125
- *IBM APPC Networking Services for Windows: Application Programmer's Reference*, SC31-8126
- *IBM APPC Networking Services for Windows: Configuration Parameters Reference for Administrators and Application Programmers*, SC31-8138

Networking Services for Windows Operating Environment

This section describes aspects of CPI Communications that are unique to IBM APPC Networking Services for Windows.

Support of CPI-C Conformance Classes

Networking Services for Windows supports the conversations class and the LU 6.2 class.

Optional Conformance Classes Supported

Networking Services for Windows supports the following optional conformance classes:

- Data conversion routines
- Full-duplex
  Networking Services for Windows uses two half-duplex, LU 6.2 conversations to transparently provide a simulated full-duplex CPI-C conversation.
- Queue-level non-blocking

© Copyright IBM Corp. 1996, 1998

- Callback function
- Security
  Networking Services for Windows supports all calls that provide security functions. However, the input parameter of CM_SECURITY_PROGRAM_STRONG on the call, Set_Conversation_Security_Type (CMSCST) is not supported.
- Server
  Networking Services for Windows supports the calls that provide the server function of accepting multiple incoming conversations. It does not support the calls that allow a program to register multiple TP names, or to manage contexts

Optional Conformance Classes Not Supported

Networking Services for Windows does not support these optional conformance classes:

- Conversation-level non-blocking — Networking Services for Windows supports queue-level non-blocking instead
- Directory
- Distributed security
- Expedited data
- OSI TP
- Recoverable transactions
- Secondary information
- Unchained transactions

Languages Supported

For information about which languages are supported by Networking Services for Windows, see Table 14 on page 108.

Pseudonym Files

Networking Services for Windows provides the pseudonym file, CPIC.H, to support the C language.

This pseudonym file is located in the INCLUDE subdirectory on the disk where Networking Services for Windows is installed. You may want to add this subdirectory to your INCLUDE environment variable.

This file contains constant declarations and data types for each supplied and returned parameter in CPI-C calls. To use another language or a non-standard implementation of a supported language, you will have to write a pseudonym file. If you are using another language, use the provided file as the model to build the pseudonym file for your language.

When writing pseudonym files, note that the Networking Services for Windows implementation of CPI-C follows the Microsoft Pascal calling conventions. In the library files, NSDW.LIB and NSDW.DLL, the symbol names corresponding to the CPI-C calls are normalized to uppercase. This is important when using a language

or compiler that is case-sensitive or when linking with the “/NOIGNORECASE” option.

CPI-C calls do not return a value; they are procedures rather than functions. The CPI-C API is invoked by far calls only. Parameters are passed by reference using far pointers pushed onto the stack in the order in which they lexically appear in the pseudonym files (*conversation_ID* first). The called CPI-C procedure removes its received parameters from the stack before returning to the caller.

Examples of Using C

One of the installation options for Networking Services for Windows is whether you want the sample programs. If you select this option, the installation program installs the sample programs and source code in the SAMPLES subdirectory. These samples are a guide to the details of compiling and linking real applications.

CPI-C Function Calls in C

The following example illustrates the Initialize_Conversation call in the C language:

```
unsigned char conversation_id[8];     /* your conversation ID      */
unsigned char sym_dest_name[8];       /* symbolic destination name */
CM_INT32 cm_retcode;                  /* return code               */

cminit(conversation_id,     /* Address of returned conversation ID     */
       sym_dest_name,       /* Address of supplied destination address */
       &cm_retcode);        /* Address of returned return code         */
```

Using the Pseudonym Files in C Language Programs

The pseudonym files contain function prototypes for each of the CPI-C library calls. These function prototypes help the compiler check for agreement of the number and type of each parameter.

The C pseudonym file, CPIC.H., contains the CPI-C constants and function prototypes.

Using Other Languages

Determine how to make calls using the Pascal calling convention in your language. The calling convention refers to the order in which procedure parameters are pushed on the stack. When multiple parameters are passed on a call, they can be pushed onto the stack in two ways:

Pascal Push the first parameter first, then the second, and so on. Your compiler would use a sequence like the following to make a CPI-C call.

```
PUSH the 32-bit address of the first parameter
PUSH the 32-bit address of the second parameter
⋮
PUSH the 32-bit address of the last parameter
CALL the 32-bit address of the specific CPI-C call
```

C Push the last parameter first, then the next-to-last, and so on.

This calling convention is not supported in Networking Services for Windows.

Linking with the CPI-C Import Library

Link your application with the CPI-C import library, NSDW.LIB, shipped with Networking Services for Windows. This library is located in the Networking Services for Windows LIB directory.

Memory Considerations

Data Buffers

For calls that require data buffers, such as Receive or Send_Data, your program can either declare the data buffer as a variable or allocate it using a library function (for example, the C `malloc()` function).

If your program specifies a buffer length greater than the actual buffer size, Networking Services for Windows might not detect this error. As a result, data lying beyond the end of the buffer might be overwritten.

Beware when allocating memory using a library function that can allocate more than 64KB (for example, the C `halloc()` function). No data buffer passed to Networking Services for Windows can be larger than 64KB.

Stack Size

Networking Services for Windows calls execute on the calling application's stack. You are responsible for ensuring that your application's stack is sufficiently large.

Defining Side Information

In Networking Services for Windows, the Initialize_Conversation call reads side information in order to find the partner LU, transaction program, and mode names associated with the specified *sym_dest_name*. For more information on creating side information for Networking Services for Windows, refer to the *IBM APPC Networking Services for Windows: Configuration Parameters Reference for Administrators and Application Programmers*.

Usage Notes for Mode_Name and TP_Name

Mode_Name

Although Networking Services for Windows allows you to set the *mode_name* characteristic to CPSVCMG or SNASVCMG using the Set_Mode_Name (CMSMN) call, it rejects an Allocate (CMALLC) call for a conversation with either of these mode names and returns a CM_PARAMETER_ERROR return code.

Specification of the blank mode for a conversation is handled in different ways depending on where the mode is set. When the mode is specified in the side information table, the string `'BLANK'` must be used. When the mode is set by means of a Set_Mode_Name (CMSMN) call, an empty string or a string containing only ASCII blanks must be specified. If you use the Extract_Mode_Name (CMEMN) call on a session using the blank mode, Networking Services for Windows returns a zero-length string.

Restrictions on Transaction Program Names

Networking Services for Windows translates transaction program names from ASCII to EBCDIC. Therefore, if a program sets the *TP_name* characteristic using a double-byte name and then calls Allocate (CMALLC), the partner LU rejects the allocation request because of an incorrect transaction program name. Transaction program names can be entered in the side information table file using a hexadecimal format specification to allow any valid SNA transaction program name. For more information about valid formats for transaction program names, refer to the *IBM APPC Networking Services for Windows: Configuration Parameters Reference for Administrators and Application Programmers*.

How Dangling Conversations Are Deallocated

When an application ends without deallocating its active conversations, those conversations are *dangling*. Networking Services for Windows automatically deallocates dangling conversations and deactivates the sessions carrying those conversations related to that application.

Dangling conversations are deallocated when the use count of the shared NSDW.DLL goes to zero. Note that the Program Launcher uses the DLL; if you want Networking Services for Windows to automatically deallocate dangling conversations, stop the Program Launcher.

Diagnosing Errors

When a Networking Services for Windows call returns an error return code, Networking Services for Windows places a message in the message log. If tracing is enabled, Networking Services for Windows also places a message in the trace log.

Log_Data

A successful call to Set_Log_Data (CMSLD) sets the *log_data* and *log_data_length* characteristics. However, log data is not sent to the conversation partner; it is only entered into the trace log when tracing is enabled.

When tracing is enabled, Networking Services for Windows places log data in the trace log when successfully entering:

- Set_Log_Data (CMSLD)
- Deallocate (CMDEAL)

Identifying Product-Specific Errors

To assist in identifying product-specific errors, enable CPI-C error tracing.

CPI-C allows a return code of CM_PRODUCT_SPECIFIC_ERROR when there is no corresponding CPI Communications return code to which the error could be mapped appropriately.

Networking Services for Windows returns the CM_PRODUCT_SPECIFIC_ERROR return code when:

- Networking Services for Windows has not been started.

- The underlying APPC implementation fails. In this case, Networking Services for Windows writes a message to the Networking Services for Windows message log. The message states that an APPC error occurred and gives the primary and secondary APPC return codes for this error.
- Some internal error occurs, such as a call failing in its attempt to allocate storage internally for its own use. In this case, Networking Services for Windows writes a message to the message log, which indicates the nature of the failure.

Deviations from the CPI Communications Architecture

Deviations from the CPI Communications Architecture are found in the following areas:

- Return_control characteristic for Allocate
- Parameter checking
- Log data support

Return_control Characteristic for Allocate (CMALLC)

If a program issues Allocate (CMALLC) with return_control set to CM_WHEN_SESSION_ALLOCATED while no sessions are available and the session limit has been reached, Networking Services for Windows does not wait for a session to become available. Instead, the CM_ALLOCATE_FAILURE_RETRY return code will be returned.

CM_PROGRAM_PARAMETER_CHECK Return Code

A Networking Services for Windows call returns CM_PROGRAM_PARAMETER_CHECK if it detects that one or more of the parameters supplied in the call are incorrect. Networking Services for Windows also places a message in the message log. If CM_PROGRAM_PARAMETER_CHECK is not returned, it does not mean that the actual parameters are valid.

Networking Services for Windows returns CM_PROGRAM_PARAMETER_CHECK if it determines that one or more of the actual parameters is a protected mode address that is not valid. Examples include selector values that specify a *conversation_ID* that cannot be read, or an output variable that cannot be written.

Log Data Support

Networking Services for Windows neither sends nor receives *log_data*. If any *log_data* is received, Networking Services for Windows discards it.

Chapter 10. CPI Communications on OS/2

This chapter summarizes the product-specific information that the OS/2 programmer needs when writing application programs that contain CPI Communications calls or OS/2 extension calls.

This chapter provides information about the Operating System/2 (OS/2) implementation of CPI Communications. CPI Communications for OS/2 is provided as part of the following products (listed in chronological order of their availability):

- Communications Manager/2 Version 1.0
- Communications Manager/2 Version 1.1
- Communications Manager/2 Version 1.11
- Communications Server

In this chapter, the term IBM Communications Server is used to refer to CPI-C function in either of the following products:

- Communications Server for OS/2 WARP Version 4.0
- Personal Communications AS/400 and 3270 for OS/2 Version 4.1

Please see Chapter 14, “CPI Communications on Win32 and 32-bit API Client Platforms” for a description of the OS/2 API Client CPI-C that is part of the following products:

- IBM eNetwork Communication Server for Windows NT 5.0, 5.01, and above
- Netware for SAA 2.2
- IntraNetware for SAA 2.3, 3.0, and above

  For the remainder of this chapter, the term *Communications Manager* collectively identifies these products. If a statement is not true for all these products, the exception is noted. The phrase *or later* is used when indicating a statement is true for a particular product and its successors (the product and the ones after it in the above list). For example, "Communications Manager/2 Version 1.1 or later" refers to the products Communications Manager/2 Version 1.1, Communications Manager/2 Version 1.11, and Communications Server.

  Communications Manager implements functions in the manner described in the main sections of this publication, except as described in “Deviations from the CPI Communications Architecture” on page 450.

  This chapter is organized as follows:

  – OS/2 Publications

  – OS/2 Operating Environment

    - Conformance Classes Supported
    - Languages Supported
    - Pseudonym Files
    - Defining Side Information
    - How Dangling Conversations Are Deallocated
    - Scope of the Conversation_ID
    - Identifying Product-Specific Errors
    - Diagnosing Errors
    - When Allocation Requests Are Sent
    - Deviations from the CPI Communications Architecture

© Copyright IBM Corp. 1996, 1998

- OS/2 Extension Calls—System Management
- OS/2 Extension Calls—Conversation
- OS/2 Extension Calls—Transaction Program Control
- OS/2 Special Notes
- Sample Program Listings for OS/2

OS/2 Publications

The following publications contain detailed product information:

- *Communications Manager/2 Version 1.0 Administration Guide*, SC31-6168
- *Communications Manager/2 Version 1.0 Application Programming Guide*, SC31-7012
- *Communications Manager/2 Version 1.0 APPC Programming Guide and Reference*, SC31-6160
- *Communications Manager/2 Version 1.0 APPN System Management Programming Reference*, SC31-6173
- *Communications Manager/2 Version 1.0 Problem Determination Reference*, SC31-6157
- *Communications Manager/2 Version 1.1 Network Administration and Subsystem Management Guide*, (SC31-6168)
- *Communications Manager/2 Version 1.1 Application Programming Guide*
- *Communications Manager/2 Version 1.1 APPC Programming Reference*
- *Communications Manager/2 Version 1.1 System Management Programming Reference*
- *Communications Manager/2 Version 1.1 Problem Determination Guide*
- *Communications Server Version 4.0 Network Administration and Subsystem Guide*, (SC31-6168)
- *Communications Server Version 4.0 Application Programming Guide and Reference*
- *Communications Server Version 4.0 APPC Programming Reference*
- *Communications Server Version 4.0 System Management Programming Reference*
- *Communications Server Version 4.0 Problem Determination Guide*

OS/2 Operating Environment

The following sections explain some special considerations that should be understood when writing applications for an OS/2 environment.

Conformance Classes Supported

Refer to "Functional Conformance Class Descriptions" on page 746 for a complete description of functional conformance classes. Also, refer to Table 15 on page 109 to determine what releases of Communications Manager support which architected CPI-C calls.

Communications Manager supports the following mandatory conformance classes:

1. Conversations

   Note: The following restrictions apply:

   - The Extract_Maximum_Buffer_Size (CMEMBS) call is only available in Communications Manager/2 Version 1.11 or later.

2. LU 6.2

In addition to the mandatory conformance classes listed above, the following optional conformance classes are available in Communications Server. They are only supported in 32-bit C Language interface and REXX.

1. Conversation-level non-blocking
2. Queue-level non-blocking
3. Server
4. Data conversion routines
5. Full-duplex conversations
6. Expedited data
7. Security - Value of CM_SECURITY_PROGRAM_STRONG is not supported.

   Also, prior to Communications Server, partial support is provided by product extension calls, refer to Table 15 on page 109 for details.
8. Secondary Information (partial support)

In addition to all the conformance classes listed above, Communications Manager supports the CPI-C extension calls documented in this chapter.

Languages Supported

The following languages can be used to issue CPI Communications calls and OS/2 extension calls:

- C
- COBOL
- FORTRAN[5]
- REXX

Note: Some of the calls are only available in Communications Server, and only at the 32-bit C Language interface and through REXX. Refer to Table 15 on page 109 for details.

---

[5] FORTRAN cannot be used to issue the following calls: Set_CPIC_Side_Information (XCMSSI), Delete_CPIC_Side_Information (XCMDSI), Extract_CPIC_Side_Information (XCMESI), and Define_TP(XCDEFTP)

The C, COBOL, and FORTRAN programming languages allow a program to include header files; the REXX language does not. The following sections list the file names for the files that contain pseudonym definitions and call prototypes. By including these files, a program can use these definitions and prototypes.

The following sections contain additional information about using individual programming languages for CPI Communications. To link CPI Communications programs written in the C, COBOL, and FORTRAN languages, define the link libraries for these languages to the system. Add the subdirectory path containing the link library to the OS/2 `SET LIB` command for the system (or environment), or prefix it to the library file name in the list of libraries on the OS/2 `LINK` statement.

Table 26. Location (OS/2 Subdirectory) of Pseudonym Files and Link Edit Files

	Communications Manager/2 Version 1.0	**Communications Manager/2 Version 1.1 (diskette)**	**Communications Manager/2 Version 1.1 Communications Server (CDROM)**
CMC.H, also CPIC.H in V2.0	\CM_H	\CM_H	\APISUPP
CMCOBOL.CBL	\CM_CBL	\CM_CBL	\APISUPP
CMFORTRN.INC	\CM_INC	\CM_INC	\APISUPP
CMREXX.CPY	\CM_CPY	\CM_CPY	\APISUPP
CPIC.LIB	\CM_LIB	\CM_LIB	\APISUPP
CPIC32.LIB			\APISUPP (Communications Server only)
CPICOBOL.LIB	\CM_LIB	\CM_LIB	\APISUPP

C

The C pseudonym file provided with Communications Manager is:

CMC.H

CPIC.H, in Communications Server, is included by CMC.H

This file contains the C pseudonym definitions, side information entry structure, and call prototypes for CPI Communications calls and OS/2 extension calls.

When compiling a C program using the IBM C/2, or C-Set++ compiler, set the warning level to 3. This level provides warning messages for mismatched data types and integer lengths. If the C pseudonym file is included in the program, and the warning level is set to 3 for compilation, the C compiler will flag calls having incorrect variables—those with a data type (integer or character) or integer length (short or long) that does not match the corresponding call prototype statement contained in the pseudonym file.

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

As part of the linkage edit step for the CPI Communications program, include the ***CPIC.LIB***, or ***CPIC32.LIB*** file in the list of libraries on the OS/2 `LINK` statement. For Communications Server, refer to the *Communications Server Version 4.0*

Application Programming Guide and Reference for information on which .LIB file to use.

COBOL

The COBOL pseudonym file provided with Communications Manager is:

CMCOBOL.CBL

This file contains the COBOL pseudonym definitions and side information entry structure for CPI Communications calls and OS/2 extension calls.

A program written to the SAA COBOL specification uses `COMP-4` integers. `COMP-4` integers in memory are in System/370 format (big-endian). Therefore, the program does not have to convert the value in the length (LL) field of a basic-conversation logical record to or from the System/370 format when using the value in an integer operation.

As part of the linkage edit step for the CPI Communications program written to the SAA COBOL specification, include the ***CPICOBOL.LIB*** file in the list of libraries on the OS/2 ***LINK*** statement.

A program can include ***COMP-5*** integers; however, this provision of the IBM COBOL/2 compiler is outside of the SAA COBOL specification, and such a program is not portable across all SAA systems. ***COMP-5*** integers in memory are in native system format (i.e. byte-reversed for OS/2). This format can improve performance if a program performs many integer operations. If the program uses ***COMP-5*** integers, it should:

- Specify the variables on the CPI Communications and Communications Manager calls in reverse order from that specified in this book.
- On the Send_Data (CMSEND) call for a basic conversation, ensure that the length (LL) value is in System/370 format before issuing the call.
- Include the ***CPIC.LIB*** file in the list of libraries on the OS/2 ***LINK*** statement, and **do not** include the ***CPICOBOL.LIB*** file.

FORTRAN

The FORTRAN pseudonym file provided with Communications Manager is:

CMFORTRN.INC

This file contains the FORTRAN pseudonym definitions and call prototypes for CPI Communications calls and OS/2 extension calls.

The following Communications Manager calls cannot be issued from a FORTRAN program:

Set_CPIC_Side_Information (XCMSSI)
Extract_CPIC_Side_Information (XCMESI)
Delete_CPIC_Side_Information (XCMDSI)

As part of the linkage edit step for the CPI Communications program written to the SAA FORTRAN specification, include the `CPIC.LIB` file in the list of libraries on the OS/2 `LINK` statement.

A program may be written that uses `EXTERNAL` statements. To do so, the program must use the `OS` prefix on the `EXTERNAL` statements. However, use of the `OS` prefix

is outside the SAA FORTRAN specification, and such a program is not portable across all SAA systems.

REXX (SAA Procedures Language)

The following sections provide information helpful in writing a REXX program for the OS/2 environment.

REXX Programs: REXX programs are really command files (also called batch files) that are executed interpretively. The file extension of .CMD must be used for REXX programs. Communications Manager supports single-threaded REXX programs.

REXX Pseudonym Definitions: The REXX pseudonym file provided with Communications Manager is:

CMREXX.CPY

This file contains the REXX pseudonym definitions for CPI Communications calls and OS/2 extension calls.

Unlike the other languages supported by Communications Manager, the REXX language does not provide an “include” capability. Therefore, the programmer may either copy the desired pseudonym definitions directly into the program, or have REXX interpret the definitions from the pseudonym file using REXX instructions for reading and interpreting lines of a file. For example:

```
fn="CMREXX.CPY"
do while lines(fn) > 0;
  INTERPRET linein(fn);
end;
```

Starting a REXX Program: The CPICREXX.EXE program must be executed before starting any CPI-C REXX program that issues CPI-C calls, since these calls use the CPICOMM environment established by CPICREXX.EXE. The CPICREXX.EXE program can be executed by the operator, from a STARTUP.CMD file, from the CONFIG.SYS file, or by any other means the user chooses, as long as it is executed before running any CPI-C REXX programs. The CPICREXX.EXE needs to be executed only once after OS/2 starts.

The CPICREXX.EXE program registers the CPICOMM environment to REXX. After the CPICOMM environment is registered, REXX programs can make CPI Communications calls.

To start a REXX program from the OS/2 command line, use the OS/2 `START` command with the `/C` command option. For example, to start a program named XCMESI.CMD from the command line, type:

```
START /C XCMESI.CMD
```

This command causes OS/2 to start a new OS/2 session for the CMD.EXE program, which then starts the REXX program. When the REXX program ends, OS/2 ends the session.

To start a REXX program by means of an inbound allocation request, configure a TP definition with CMD.EXE as the file name and the REXX program file name as the parameter string for the TP definition. Precede the file name in the parameter

string with the /C option. For example, to start the program named XCMSSI.CMD by means of an inbound allocation request, configure the parameter string for the TP definition as:

```
/C XCMSSI.CMD
```

This configuration causes OS/2 to start a new OS/2 session for the CMD.EXE program, which then starts the REXX program. When the REXX program ends, OS/2 ends the session.

Note: A program that is started manually from the OS/2 command line is referred to as an operator-started program. One that is started by an inbound allocation request is referred to as an "attach manager started" program.

Issuing a Call Using REXX: To issue a call using the REXX language, type:

```
ADDRESS CPICOMM 'callname variable0 variable1 ... variableN'
```

CPICOMM is the environment name the program uses to invoke CPI Communications calls. The program passes the call name and variable names as a character string to the CPICOMM environment. In Communications Manager, the CPICOMM environment is registered to REXX by the CPICREXX.EXE program before running any REXX CPI Communications programs.

Some character-string variables used on the OS/2 extension calls do not have an associated length variable. For example:

key
sym_dest_name

For languages other than REXX, these variables must be at least 8 bytes long. REXX programs can specify variables that are shorter than 8 bytes—that is, REXX variables have the same length as the character string they contain.

A similar exception applies to the elements of REXX arrays such as the array used for the *side_info_entry* fields and the (XCDEFTP) *TP_definition* fields. REXX programs can specify array elements that are shorter than the corresponding field lengths—elements having the same length as the character string they contain. Conversely, REXX programs can specify array elements that are longer than the corresponding *side_info_entry* field lengths. However, Communications Manager ignores any characters of an array element that are beyond the length of the corresponding field.

Checking the REXX Return Code: The CPICOMM environment uses the call name and variable names to create the actual call. In doing so, it can encounter certain error conditions prior to issuing the call. If it encounters an error, it returns to the REXX program without issuing the CPI Communications call. It sets the REXX return code variable, RC, to a non-zero value when it encounters an error. The value may be negative or positive, depending on the error. If it does not encounter any errors, the CPICOMM environment issues the call and, upon completion of the call, sets the REXX return code variable to zero.

The return code values that the CPICOMM environment can return in the RC variable are shown in Table 27.

Table 27. Values Returned in the REXX RC Variable for Communications Manager	
RC Value	Meaning
0	The ADDRESS CPICOMM statement completed with no REXX errors.
+30	The CPICREXX.EXE program has not been executed.
−3	The CPICOMM environment does not recognize the call name specified on the ADDRESS CPICOMM statement.
−9	The CPICOMM environment requested a buffer from the CPI Communications component of Communications Manager to create the call, but insufficient buffers were available.
−10	The REXX program supplied too many variable names for the call.
−11	The REXX program supplied too few variable names for the call.
−24	The CPICOMM environment encountered a REXX fetch failure; this usually means the REXX program supplied a name for a variable that does not exist.
−25	The CPICOMM environment encountered a REXX set failure; this means REXX memory has been exhausted.
−28	The REXX program supplied an invalid variable name.

Following the call, the REXX program should check the RC variable before it processes any values returned in the CPI Communications variables, including the CPI Communications return code. If the RC value is anything other than 0, the output parameters of the call are not meaningful.

Ending a REXX Program: For Communications Manager to know when a REXX program ends, the OS/2 session in which the CMD.EXE program is running must end. When the session ends, control transfers to a Communications Manager exit routine to perform cleanup processing. If the session does not end, Communications Manager does not get control when the program is finished, and resources such as the TP instance and dangling conversations cannot end. This might require operator intervention to deactivate the sessions allocated to the conversations, to free the session resources for subsequent use.

Following are some examples of how to cause the OS/2 session to end when the REXX program is finished:

- Use the /C option on the OS/2 START command for operator-started programs, or precede the program file name with /C on the parameter string for the TP definition.
- End the REXX program with one of the following EXIT statements:

```
ADDRESS CMD 'EXIT'
'EXIT'
```

  The first statement explicitly addresses the EXIT to the CMD environment, which causes the CMD.EXE session to end. The second statement is equivalent to the first and can be used to end the OS/2 session if the current environment is the CMD environment—that is, if the program has not permanently changed the environment from the time it started.

- For operator-started programs, end the session by entering the OS/2 EXIT command.

Defining or Referencing a Side Information Entry using REXX: The Set_CPIC_Side_Information (XCMSSI) and Extract_CPIC_Side_Information (XCMESI) calls supply a data structure as one of the variables. REXX programs cannot create the data structure as shown in the description of these calls. For a REXX program to issue these calls, it must do the following:

1. Use the REXX array capability to create the structure.
2. The name for the array, such as *sideinfo*, may be chosen by the program.
3. Each element of the array must consist of the array name and the *side_info_entry* field name as the array element name. For example, the first field of a side information entry is the *sym_dest_name*, so the first element of the *sideinfo* array would be named *sideinfo.sym_dest_name*
4. On the REXX statement for the call, supply the array name as the *side_info_entry* variable name—*sideinfo* in this example.

Defining or Referencing TP_Definition Using REXX: The same type of considerations for defining or referencing side information fields used on the XCMSSI call (see “Defining or Referencing a Side Information Entry using REXX”) are applicable for defining or referencing the TP_Definition structure using REXX.

Pseudonym Files

Integer characteristics, variables, and fields are shown throughout this chapter as having pseudonym values rather than integer values. For example, instead of stating that the variable *conversation_security_type* is set to an integer value of 0, this chapter shows *conversation_security_type* set to the pseudonym value of CM_SECURITY_NONE. In addition to the pseudonyms used with base CPI Communications, Communications Manager provides pseudonyms for additional characteristics, fields, and variables.

For the file name that contains the pseudonym definitions for each supported language, see “Languages Supported” on page 437.

Defining Side Information

The set of parameters associated with a given symbolic destination name is called a side information entry. This section provides an overview of how a Communications Manager user or program can add, replace, delete, and extract side information entries.

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

User-Defined Side Information

The Communications Manager configuration panels can be used to create and update side information entries in a configuration file. In addition, if the configuration file is active, Communications Manager can be requested to also update the side information in its internal memory. In general, using the configuration panels requires an operator, a keyboard, and a display—a typical, if not universal, Communications Manager environment.

The configuration file can also be updated using an editor. This method is useful, for example, when it is necessary to configure side information for multiple systems and distribute the configuration file among the systems.

For information about all of the Communications Manager configuration capabilities, including details about how a user can configure CPI Communications side information, refer to the *Communications Manager/2 Configuration Guide*.

After the configuration file is updated, Communications Manager verification of the configuration file may be initiated. As an option, one can request that changes made to the side information in the file also take effect in the internal side information, provided that the configuration file is active and the verification is successful.

Program-Defined Side Information

The programmer can write a system management program that issues Communications Manager calls to update the internal side information entries and obtain the parameter values of the entries. These updates affect only the internal side information and remain in effect until changed or until Communications Manager is stopped; they do not alter the configuration file. Similarly, parameter values are obtained only from the internal side information; no reference is made to the configuration file. Consequently, these calls are useful for making temporary updates to the internal side information without affecting the original information in the configuration file.

Communications Manager provides calls to update the internal side information and one to extract it. These are:

- Set_CPIC_Side_Information (XCMSSI)
  Add or replace the entry (all parameter values) for a symbolic destination name. See page 607 for a detailed description.
- Delete_CPIC_Side_Information (XCMDSI)
  Delete the entry for a symbolic destination name. See page 602 for a detailed description.
- Extract_CPIC_Side_Information (XCMESI)
  Return the entry for a symbolic destination name or for the *n*th entry. See page 457 for a detailed description.

Note: For REXX programs, refer to “Defining or Referencing a Side Information Entry using REXX” on page 443 for information on how to define side information entries.

The symbolic destination name provides the index on each of these calls for accessing a side information entry. Alternatively, an integer value provides the index on the Extract_CPIC_Side_Information call for extracting an entry. If the program calls Set_CPIC_Side_Information and an entry for the symbolic destination

name does not exist, Communications Manager adds a new entry. If the entry does exist, Communications Manager replaces all of the entry's parameters. When the program calls Delete_CPIC_Side_Information, Communications Manager removes the entire entry (the symbolic destination name and all of its associated parameters). If the program calls Extract_CPIC_Side_Information, all parameters for the entry are returned except *security_password*.

The Communications Manager keylock feature is used with the Set_CPIC_Side_Information and Delete_CPIC_Side_Information calls. The feature may be enabled (secured) during Communications Manager configuration. This feature provides a means for protecting a system against unauthorized change to Communications Manager system definition parameters, including CPI Communications side information. When the keylock feature is secure, a program can issue the Set_CPIC_Side_Information and Delete_CPIC_Side_Information calls only if it supplies the Communications Manager master or service key. Refer to the *Communications Manager/2 Configuration Guide* for details about the keylock feature.

Side Information Parameters

Table 28 shows the parameters contained in a Communications Manager CPI Communications side information entry and a brief description of each parameter. Refer to “Characteristics, Fields, and Variables” on page 631 for a definition of the data type and length of these parameters.

Table 28 (Page 1 of 2). An Entry of Communications Manager CPI Communications Side Information

Parameter Pseudonym	Description
sym_dest_name	The name a program specifies on Initialize_Conversation to assign the initial characteristic values for the conversation.
partner_lu_name	The alias or network name of the partner LU for the conversation.
tp_name_type	An indicator of whether the *tp_name* parameter specifies an application TP name or an SNA service TP name. The value may be: • XC_APPLICATION_TP • XC_SNA_SERVICE_TP
tp_name	The name of the partner program for the conversation. The program may be an application TP or an SNA service TP.
mode_name	The name of the session mode for the conversation.

Table 28 (Page 2 of 2). An Entry of Communications Manager CPI Communications Side Information

Parameter Pseudonym	Description
conversation_security_type	The level of access security information to include on the allocation request sent to the partner LU. This parameter is defined only from the configuration panels or by a program call to Set_CPIC_Side_Information; it is not included in the editable configuration file. The value may be: • CM_SECURITY_NONE (or XC_SECURITY_NONE) • CM_SECURITY_SAME (or XC_SECURITY_SAME) • CM_SECURITY_PROGRAM (or XC_SECURITY_PROGRAM) See "Set_Conversation_Security_Type (XCSCST)" on page 616 for a description of these values.
security_user_ID	The access security user ID to include on the allocation request sent to the partner LU, when *conversation_security_type* is CM_SECURITY_PROGRAM (or XC_SECURITY_PROGRAM). When *conversation_security_type* is other than CM_SECURITY_PROGRAM (or XC_SECURITY_PROGRAM), this parameter is ignored. This parameter is defined only from the configuration panels or by a program call to Set_CPIC_Side_Information; it is not included in the editable configuration file.
security_password	The access security password to include on the allocation request sent to the partner LU, when *conversation_security_type* is CM_SECURITY_PROGRAM (or XC_SECURITY_PROGRAM). When *conversation_security_type* is any value other than CM_SECURITY_PROGRAM (or XC_SECURITY_PROGRAM), this parameter is ignored. This parameter is defined only from the configuration panels or by a program call to Set_CPIC_Side_Information; it is not included in the editable configuration file.

How Dangling Conversations Are Deallocated

When a CPI Communications program starts, Communications Manager adds itself to an OS/2 exit list for the program's OS/2 process. When the program ends its execution, OS/2 gives control to the Communications Manager exit routine. The Communications Manager exit routine performs cleanup processing on behalf of the program.

With one exception (described in the following paragraph), Communications Manager deallocates dangling conversations, as part of its cleanup processing. That is, it deallocates all remaining conversations for that program, using the *deallocate_type* of CM_DEALLOCATE_ABEND.

The exception to this deallocation is a program written in the REXX language. Communications Manager may not get control when a REXX program ends. Therefore, REXX programs should deallocate all conversations before ending. Failure to do so may require operator intervention to deactivate the sessions allocated to the conversations, to free the session resources for subsequent use. See “Ending a REXX Program” on page 442 for information about how to end the execution of a REXX program.

It is a good practice for all CPI Communications programs to deallocate all active conversations when they are finished with them. And, of course, programs that require their conversations to be deallocated with a *deallocate_type* other than CM_DEALLOCATE_ABEND must deallocate them before ending execution.

Scope of the Conversation_ID

The scope of the *conversation_ID* in OS/2 is limited to one TP instance—that is, the TP instance with which it was associated when the conversation was initialized. For more information, refer to “TP Instances for Communications Manager” on page 625.

Identifying Product-Specific Errors

When CPI Communications returns the CM_PRODUCT_SPECIFIC_ERROR return code, it creates an entry in the error log. Information in the error log entry identifies CPIC as the originator. Refer to the problem determination guide for the specific product being used for a complete description of the errors and the recommended action to take.

Notes:

1. The state of the conversation remains unchanged, except for conditions where APPC is stopped or ends abnormally.
2. As part of its processing of a CPI Communications call other than a Set or Extract call, the CPI Communications component of Communications Manager creates an APPC verb and calls the APPC component to execute the verb. For example, when a program calls Allocate (CMALLC) for a basic conversation, Communications Manager creates and executes the APPC verb, ALLOCATE. When the APPC component of Communications Manager encounters an unexpected error on an OS/2 call, it logs the error as type 0022. When the APPC component returns to the CPI Communications component, the latter logs an additional entry as type 0052. The second log entry indicates that the error occurred while Communications Manager was processing a CPI Communications call.

3. If Communications Manager is not active when the program issues the Initialize_Conversation or Accept_Conversation call, the CM_PRODUCT_SPECIFIC_ERROR return code is returned on the call but no error is logged.

Diagnosing Errors

This section provides information on diagnosing errors. Refer to the problem determination guide for the specific product being used for an additional description of the errors and the recommended action to take. Refer also to the CPI Communications return codes given in *Appendix B, "Return Codes and Secondary Information"* and Table 27 on page 442.

Set_Log_Data (CMSLD)

The program sets the *log_data* characteristic to a character string representing the message text portion of the Error Log GDS variable. The *log_data* variable on the CMSLD call must contain a 1–512 byte ASCII character string, and it may include the space character. Communications Manager translates all *log_data* characters to EBCDIC before recording the log data in the system error log and sending the Error Log GDS variable on the conversation. The log data is written to the error log with a type of 0001 and a subtype set to the sense data Communications Manager sends with the Error Log GDS variable on the conversation.

Logging Errors for CPI Communications Error Return Codes

Communications Manager provides the user with a means for identifying errors and return codes. Communications Manager stores the errors in the OS/2 system error log. Refer to the problem determination guide for the specific product being used for a complete description of the errors and the recommended action to take.

Communications Manager creates an error log entry, as discussed in "Identifying Product-Specific Errors" on page 447, when it logs the CM_PRODUCT_SPECIFIC_ERROR.

Communications Manager creates an error log entry for session activation failures and session outages. If the CPI Communications program receives one of the following error return codes, Communications Manager may also record a system error log entry:

CM_ALLOCATE_FAILURE_NO_RETRY
CM_ALLOCATE_FAILURE_RETRY
CM_RESOURCE_FAILURE_NO_RETRY
CM_RESOURCE_FAILURE_RETRY

The first two return codes are indicated on an Allocate (CMALLC) call when Communications Manager fails to allocate a session. The last two return codes are indicated on calls following an Allocate or Accept_Conversation (CMACCP) call when Communications Manager detects a session outage. Communications Manager logs session outages in all cases. However, it logs allocation failures only when the cause is a session activation failure. Allocation failures caused by the session limit being 0 for the session mode name are not logged.

Session activation errors and session outages can be caused by a number of different conditions. The system error log specifies a type and subtype combination for each condition.

Communications Manager creates an error log entry for error information sent by the partner program. These error log entries all have a type of 0001 and a subtype set to the sense data Communications Manager receives with the error notification. The CPI Communications return codes for these conditions are:

CM_DEALLOCATE_ABEND
CM_PROGRAM_ERROR_NO_TRUNC
CM_PROGRAM_ERROR_PURGING
CM_PROGRAM_ERROR_TRUNC
CM_SVC_ERROR_NO_TRUNC
CM_SVC_ERROR_PURGING
CM_SVC_ERROR_TRUNC

Causes for the CM_PROGRAM_PARAMETER_CHECK Return Code

This section discusses the causes for the CM_PROGRAM_PARAMETER_CHECK return code that are specific to Communications Manager. These causes are in addition to those described in *Appendix B*, "Return Codes and Secondary Information." In Communications Server, CM_PROGRAM_PARAMETER_CHECK return codes usually cause additional information to be written to the Communications Manager message log.

Communications Manager indicates the CM_PROGRAM_PARAMETER_CHECK return code because:

- The call passed a pointer to a variable and the pointer is not valid. An invalid pointer contains a memory address that Communications Manager cannot use to refer to the variable. Examples of a invalid pointer are:
  - An address within a code segment
  - An address greater than the data segment limit
  - An address of zero (which is called a null address)
- The call passed a pointer to a character string variable (including a buffer variable) and a length, but the length would extend the valid addressability for the data beyond the segment limit.
- For non-blocking operations that initially go outstanding (CM_OPERATION_INCOMPLETE return code), if a return parameter becomes inacessible, then the CM_PROGRAM_PARAMETER_CHECK return code will be returned. This could be caused, for example, if a C Language program freed the storage by returning it to OS/2.

Causes for the CM_PROGRAM_STATE_CHECK Return Code

This section discusses the causes for the CM_PROGRAM_STATE_CHECK return code on the Accept_Conversation (CMACCP) call that are specific to Communications Manager. These causes are in addition to those described for the base CPI Communications call, as given in *Appendix B, "Return Codes and Secondary Information"*. Communications Manager does not log these errors.

Communications Manager indicates the CM_PROGRAM_STATE_CHECK return code on the Accept_Conversation call because:

- The operator or program set a TP name in the APPCTPN environment variable that was incorrect; that is, it did not match the TP name on the inbound allocation request for the program.

- An operator-started program issued the Accept_Conversation call, but the call expired before the inbound allocation request arrived. The duration that a call waits for an inbound allocation request is configured on the TP definition, using the `receive_allocate_timeout` parameter.
- For an Accept_Conversation or Accept_Incoming call, at least one TP name being used by the call is a Communications Manager non-queued TP and there are no outstanding attaches matching any of the TP name(s) being waited on.
- An operator-started or attach manager-started program issued the Accept_Conversation call after the inbound allocation request for the program expired. The duration that an inbound allocation request waits for an Accept_Conversation call is configured on the TP definition, using the `incoming_allocate_timeout` parameter.

When Allocation Requests Are Sent

Because Communications Manager buffers data being transmitted to the remote LU, the allocation request generated by an Allocate call is not sent to the remote node until one of the following occurs:

- The local LU's send buffer becomes full as the result of (one or more) Send_Data calls.
- A call is executed that explicitly flushes the buffer (for example, a Flush call or a Receive call).

Deviations from the CPI Communications Architecture

OS/2 supports CPI Communications calls except as indicated in the following sections.

Accept_Incoming (CMACCI)

When an Accept_Incoming (CMACCI) call is issued to Communications Manager, the following rules are checked sequentially to determine if it applies. The TP name(s) is determined by the first applicable rule.

1. The TP name from a successfully completed Set_TP_Name (CMSTPN) call for this conversation.

   This rule is only used for Communications Server.

2. The TP name(s) from a successfully completed Specify_Local_TP_Name (CMSLTP) call for this conversation. Refer to "Specify_Local_TP_Name (CMSLTP)" on page 451 for allowable TP names on the CMSLTP call.

3. A TP name set in the `APPCTPN` OS/2 environment variable. The environment variable must be set to a valid TP name.

   Partially specified TP names and '*' (allowed on the CMSLTP call) are not allowable TP names for the `APPCTPN` environment variable.

If none of the above rules yields a TP name, then CMACCI completes with a CM_PROGRAM_STATE_CHECK return code.

Release_Local_TP_Name (CMRLTP)

In addition to the TP names described in “Specify_Local_TP_Name (CMSLTP),” Communications Manager allows the TP name XC_RELEASE_ALL on the CMRLTP. Use of this name will result in releasing all of the TP names for this program.

If you are writing a multi-threaded CPI-C program, refer to “Multi-threaded CPI-C Programs” on page 624.

Specify_Local_TP_Name (CMSLTP)

Communications Manager allows the following TP names on the CMSLTP call:

- A fully specified name.
  The allowable characters and length are specified in Table 40 on page 632 For example: MYTP12.
- A partially specified name.
  This option allows CPI-C to accept conversations for TP names that begin with a partially specified name. For example: MYTP* will accept conversations for TP names MYTP1, MYTPFROMHEADQUARTERS, etc.
- '*'
  This option will accept any LU 6.2 incoming conversations for this entire workstation (instance of Communications Manager). This option should be used very judiciously and only if this program is the only LU 6.2 program running under the entire OS/2 operating system.

CPI-C issues the RECEIVE_ALLOCATE(multiple) APPC verb when processing the CMACCI call. If a TP name for an incoming conversation has not been configured in Communications Manager, there are cases where Communications Manager will dynamically define the TP characteristics when using the above options. Refer to RECEIVE_ALLOCATE(multiple) call in the *Communications Manager/2 APPC Programming Reference* for details.

Duplicate CMSLTP entries are not removed by Communications Manager. For example, if CMSLTP is issued twice and in each case the TP name MYTP1 is specified, then the list of local TP names will contain two MYTP1 entries. A CMRLTP call specifying a TP name of MYTP1 must be issued twice to removed these entries.

If you are writing a multi-threaded CPI-C program, refer to “Multi-threaded CPI-C Programs” on page 624.

Set_Sync_Level (CMSSL)

Communications Manager supports the following synchronization levels:

CM_NONE
CM_CONFIRM

If the program specifies other values (than those listed above) on the *sync_level* variable, Communications Manager rejects the Set_Sync_Level call with a *return_code* of:

- CM_PARM_VALUE_NOT_SUPPORTED for Communications Manager/2 Version 1.11 or later
- CM_PROGRAM_PARAMETER_CHECK for "Communications Manager/2 Version 1.0 and Communications Manager/2 Version 1.1

Programming Languages Not Supported

CPI Communications calls from Application Generator, PL/I, and RPG programs are not supported on OS/2 systems.

Mode Names Not Supported

Communications Manager does not allow a CPI Communications program to allocate a conversation that uses the CPSVCMG or SNASVCMG mode name. If the program attempts to call Allocate (CMALLC) with the *mode_name* characteristic set to CPSVCMG or SNASVCMG, Communications Manager rejects the call with a *return_code* of CM_PARAMETER_ERROR.

CPI Communications Functions Not Available

This section lists the CPI Communications functions that are not available at the CPI Communications interface on Communications Manager.

Unsupported TP Names: Communications Manager does not support double-byte TP names and provides limited support for SNA service TP names.

Double-Byte TP Names: Communications Manager does not support TP names consisting of characters from a double-byte character set, such as Kanji. These TP names begin with the X'0E' character and end with the X'0F' character. They have an even number of bytes (2 or more) between these delimiting characters.

If the program calls Allocate (CMALLC) with the *TP_name* characteristic set to a double-byte name, Communications Manager treats the name as ASCII and translates each byte to EBCDIC. The resulting TP name is not valid, and the partner LU for the conversation rejects the allocation request.

SNA Service TP Names: Communications Manager does not support the specification of an SNA service TP name on the Set_TP_Name (CMSTPN) call, nor does it support the setting of the APPCTPN OS/2 environment variable to an SNA service TP name.

Specifically, a TP name can be supplied in the following ways:

- Specified on the Specify_Local_TP_Name (CMSLTP) call
- Specified on the Set_TP_Name (CMSTPN) call
- Specified in the side information—by means of either the configuration panels or the Set_CPIC_Side_Information (XCMSSI) call
- Set in the APPCTPN OS/2 environment variable for operator-started programs.

A TP name specified on the CMSTPN call or in the side information is used when a program issues the Allocate (CMALLC) call for an outbound conversation. A TP name set in the APPCTPN OS/2 environment variable or specified on the CMSLTP call is used when an operator-started program issues the Accept_Conversation (CMACCP) call or Accept_Incoming (CMACCI) call for an inbound conversation. Communications Manager treats a TP name specified on the CMSTPN call or a TP name set in the APPCTPN OS/2 environment variable as an application TP name and translates all characters of the name from ASCII-to-EBCDIC before using the name.

However, Communications Manager is able to distinguish an application TP name from an SNA service TP name specified in the side information. The side information includes a flag, *TP_name_type*, that indicates whether the TP name is

an application TP name or an SNA service TP name. Communications Manager translates all characters of an SNA service TP name (except the first character) from ASCII-to-EBCDIC before using the name.

Remotely started CPI Communications programs—local programs that are started by inbound allocation requests—cannot be SNA service TPs. That is, Communications Manager cannot complete a CMACCP or CMACCI call for an inbound allocation request that specifies an SNA service TP name. This restriction exists because Communications Manager sets the TP name (converted from EBCDIC-to-ASCII) in the APPCTPN OS/2 environment variable for the program. Then, when the program issues the CMACCP or CMACCI call, Communications Manager retrieves the TP name from the environment variable to complete the call, just as it does for an operator-started program. However, Communications Manager sets only application TP names in the environment variable, not SNA service TP names (because of the conflict with ASCII characters and the first character of an SNA service TP name). If the inbound allocation request specifies an SNA service TP name, the CPI Communications program's CMACCP (or CMACCI) fails with a *return_code* of CM_PRODUCT_SPECIFIC_ERROR.

OS/2 Extension Calls—System Management

A program issues system management calls to do the following:

- Set, delete, or extract parameter values as defined in the Communications Manager internal side information. The side information is used to assign initial characteristic values on the Initialize_Conversation (CMINIT) call. The program can also extract the current values of the side information.
- Define or delete TP definitions.
- Register or unregister memory objects.

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

Table 29 lists the system management call names and briefly describes their functions.

Table 29. List of Communications Manager System Management Calls

Call	Pseudonym	Description
XCMDSI	Delete_CPIC_Side_Information	Deletes a side information entry for a specified symbolic destination name.
XCMESI	Extract_CPIC_Side_Information	Returns the parameter values of a side information entry for a specified symbolic destination name or entry number.
XCMSSI	Set_CPIC_Side_Information	Sets the parameter values of a side information entry for a specified symbolic destination name.
XCDEFTP	Define_TP	Defines a TP.
XCDELTP	Delete_TP	Deletes a TP definition.
XCRMO	Register_Memory_Object	Registers a memory object.
XCURMO	Unregister_Memory_Object	Unregisters a memory object.

Delete_CPIC_Side_Information (XCMDSI)

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

See "Defining Side Information" on page 443 for more information about configuring side information.

Format

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

Parameters

key *(input)*

Specifies either the master or service key, when the Communications Manager keylock feature has been secured. The use of the *key* deters unintentional or unauthorized changes to the side information. See the *Communications Manager/2 Configuration Guide* for the product being used for details of the keylock feature.

If keylock is not secured, the value of this parameter is ignored.

sym_dest_name *(input)*

Specifies the symbolic destination name for the side information entry to be removed.

return_code *(output)*

Specifies the result of the call execution. The *return_code* can be one of the following:

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

State Changes

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

Usage Notes

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

Extract_CPIC_Side_Information (XCMESI)

A program issues the Extract_CPIC_Side_Information (XCMESI) call to obtain the parameter values of an entry in Communications Manager internal side information. The program requests the entry by either an entry number or a symbolic destination name. It does not access side information in the configuration file.

See "Defining Side Information" on page 443 for information about configuring side information.

Format

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

Parameters

entry_number *(input)*

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

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

sym_dest_name *(input)*

Specifies the symbolic destination name of the entry, when parameter values for a named entry are needed. Communications Manager uses this variable only when *entry_number* is 0. If *entry_number* is greater than 0, Communications Manager ignores this *sym_dest_name* variable.

side_info_entry *(output)*

Specifies a structure in which the parameter values are returned. The format of the structure is shown in Table 30. Values within character string fields are returned left-justified and padded on the right with space characters.

Table 30. Entry Structure for the Communications Manager Extract_CPIC_Side_Information Call

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

The following extended structure is available in Communications Server to support 10-byte user_IDs and 10-byte user_passwords:

Table 31. Extended Entry Structure for the Communications Server Call

Byte Offset	Field Length and Type	Parameter Pseudonym
0	8-byte character string	*sym_dest_name*
8	17-byte character string	*partner_lu_name*
25	3-byte character string	(reserved)
28	32-bit integer	*tp_name_type*
32	64-byte character string	*tp_name*
96	8-byte character string	*mode_name*
104	32-bit integer	*conversation_security_type*
108	10-byte character string	*security_user_ID*
118	22-byte character string (must be zeros)	(reserved)

Refer to Table 28 on page 445 for descriptions of the side information parameters. Refer to Table 40 on page 632 for definitions of the character set usage and length of each character string parameter.

side_info_entry_length *(input)*
Specifies the length of the entry structure. Set this length to 124, or 140 for Communications Server if using the extended side information stucture.

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

- CM_OK

Successful completion.

- CM_PROGRAM_PARAMETER_CHECK
  This return code indicates one of the following:
  - The *entry_number* specifies a value greater than the number of entries in the side information.
  - The *entry_number* specifies a value less than 0.
  - The *sym_dest_name* specifies a name that is not in any entry in the internal side information, and *entry_number* specifies 0.
  - The *side_info_entry_length* contains a value other than 124 or 140.
  - The address of one of the variables is not valid.
- CM_PRODUCT_SPECIFIC_ERROR
  The APPC component of Communications Manager is not active for one of the following reasons:
  - The Communications Manager has not started APPC.
  - The Communications Manager has stopped APPC.
  - APPC is in an abend state.

State Changes

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

Usage Notes

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

   C
   COBOL
   REXX

   Refer to "Languages Supported" on page 437 for information on how to create the data structure using these languages.
8. The Extract_CPIC_Side_Information call and the Set_CPIC_Side_Information (XCMSSI) call use the same *side_info_entry* format. This format enables a program to obtain an entry, update a field, and restore the updated entry, provided the entry contains no *security_password*, or the program also updates the *security_password*.
9. The *entry_number* specifies an index into the current list of internal side information entries. If entries are deleted, the indexes for particular entries may change.

Set_CPIC_Side_Information (XCMSSI)

A program issues the Set_CPIC_Side_Information (XCMSSI) call to add or replace an entry in Communications Manager internal side information. The entry contains all the side information parameters for the conversation identified by the supplied symbolic destination name. If the entry does not exist in the side information, this call adds a new entry; otherwise, it replaces the existing entry in its entirety.

Side information in the configuration file remains unchanged. This call overrides the side information copied from the active configuration file when Communications Manager was started.

See "Defining Side Information" on page 443 for more information about configuring side information.

Format

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

Parameters

key *(input)*

Specifies either the master or service key, when the Communications Manager keylock feature has been secured. The use of the key deters unintentional or unauthorized changes to the side information. See the *Communications Manager/2 Configuration Guide* for the product being used for details of the keylock feature.

If keylock is not secured, the value of this parameter is ignored.

side_info_entry *(input)*

Specifies the structure containing the parameter values for the side information entry. The format of the structure is shown in Table 32. Values within character string fields must be left-justified and padded on the right with space characters.

Table 32 (Page 1 of 2). Entry Structure for the Communications Manager Set_CPIC_Side_Information Call

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

Table 32 (Page 2 of 2). Entry Structure for the Communications Manager Set_CPIC_Side_Information Call

Byte Offset	Field Length and Type	Parameter Pseudonym
116	8-byte character string	*security_password*

The following extended structure is available in Communications Server to support 10-byte user_IDs and 10-byte user_passwords:

Table 33. Extended Entry Structure for the Communications Server Call

Byte Offset	Field Length and Type	Parameter Pseudonym
0	8-byte character string	*sym_dest_name*
8	17-byte character string	*partner_LU_name*
25	3-byte character string	(reserved)
28	32-bit integer	*TP_name_type*
32	64-byte character string	*TP_name*
96	8-byte character string	*mode_name*
104	32-bit integer	*conversation_security_type*
108	10-byte character string	*security_user_ID*
118	10yte character string	*security_password*
128	12-byte character string (must be zeroes)	(reserved)

Refer to Table 28 on page 445 for a description of the side information parameters; refer to Table 40 on page 632 for a definition of the character set usage and the length of each character string parameter.

side_info_entry_length *(input)*
Specifies the length of the entry structure. Set this length to 124, or 140 for Communications Server if using the extended side information stucture.

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

- CM_OK
  Successful completion.
- CM_PROGRAM_PARAMETER_CHECK
  This return code indicates one of the following:
  - The *key* variable contains a value that does not match the master or service key and the Communications Manager keylock feature has been secured.
  - The *sym_dest_name* field in the *side_info_entry* structure contains a space character in the left-most byte (at byte offset 0 of the structure).
  - The *tp_name_type* field in the *side_info_entry* structure specifies an undefined value.

  - The *conversation_security_type* field in the *side_info_entry* structure specifies an undefined value.
  - The *side_info_entry_length* contains a value other than 124 or 140.
  - The address of one of the variables is not valid.
- CM_PRODUCT_SPECIFIC_ERROR
  The APPC component of Communications Manager is not active for one of these reasons:
  - The Communications Manager has not started APPC.
  - The Communications Manager has stopped APPC.
  - APPC is in an abend state.

State Changes

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

Usage Notes

1. If a *return_code* other than CM_OK is returned on this call, the side information entry is not created or changed.
2. The character string parameter values supplied in the fields of the *side_info_entry* structure are not checked for validity on this call. An invalid parameter value is detected later on the Allocate (CMALLC) call or on a subsequent call, such as Send_Data (CMSEND), depending on which parameter value is not valid. An invalid partner LU name or mode name is detected on the Allocate call and indicated to the program on that call. The partner LU detects an invalid TP name, user ID, or password when it receives the allocation request; in this case the partner LU returns an error indication to the program on a subsequent call following the Allocate.
3. This Set_CPIC_Side_Information call does not affect any active conversation.
4. The side information supplied on this call takes effect immediately and is used for all Initialize_Conversation calls for the new or changed symbolic destination name made after completion of this call.
5. While Communications Manager updates the side information with the parameters from this call, any other program's call to change or extract the side information is suspended until this call is completed; this includes a program's call to Initialize_Conversation (CMINIT).
6. The Set_CPIC_Side_Information call is available on Communications Manager for the following SAA programming languages:
   - C
   - COBOL
   - REXX

   Refer to “Languages Supported” on page 437 for information on how to create the data structure using these languages.
7. The Set_CPIC_Side_Information call and the Extract_CPIC_Side_Information (XCMESI) call use the same *side_info_entry* format. This format enables a program to obtain an entry, update a field, and restore the updated entry, provided the entry contains no *security_password*, or the program also updates the *security_password*.

Define_TP (XCDEFTP)

This call is only available in Communications Server, and only at the 32-bit C Language interface and through REXX.

A program uses the Define_TP (XCDEFTP) call to define information that affects the way Communications Manager processes an incoming request for a conversation with the specified transaction program (TP) name. This TP name can subsequently be associated with a running program via the Specify_Local_TP_Name (CMSLTP) call. The definition information includes the characteristics of conversations that the TP can accept, the file specification of the local program, and the timeout values for both incoming conversation requests and accept calls.

The XCDEFTP call can be used to add a new TP definition or to replace an existing one. If the TP is already defined but has not been associated with a program with a CMSLTP call, the existing definition is replaced. If the TP name is already associated with a program, the call is rejected.

A TP definition can be removed with a Delete_TP (XCDELTP) call.

More information concerning XCDEFTP and XCDELTP can be found in the Define_TP and Delete_TP verb sections of the *Communications Server Version 4.0 System Management Programming Reference*.

Format

```
CALL XCDEFTP(key,
             TP_definition,
             TP_definition_length,
             return_code)
```

Parameters

key *(input)*
: Specifies either the master or service key, when the Communications Manager keylock feature has been secured. See the *Communications Server Installation and Configuration Guide* for details of the keylock feature.

  This is an 8-byte character field. If the key is shorter than 8 characters, it must be left-justified and padded on the right with space characters.

TP_definition *(input)*
: Specifies the structure containing the parameter values for the TP definition entry. The format of the structure is shown in Table 34 on page 464. Values within character string fields must be left-justified and padded on the right with space characters.

Table 34. Entry Structure for the Communications Server Define_TP Call.		
Byte Offset	**Field Length and Type**	**Parameter Pseudonym**
0	64-byte character string (ASCII)	TP_name
64	32-bit integer	TP_name_type (XC_APPLICATION_TP, XC_SNA_SERVICE_TP)
68	80-byte character string (ASCII)	filespec
148	80-byte character string (ASCII)	icon_filespec
228	128-byte character string (ASCII)	parm_string
356	32-bit integer	conversation_type (CM_BASIC_CONVERSATION, CM_MAPPED_CONVERSATION, XC_EITHER_CONVERSATION_TYPE)
360	32-bit integer	send_receive_mode CM_HALF_DUPLEX, CM_FULL_DUPLEX, XC_EITHER_SEND_RECEIVE_MODE)
364	32-bit integer (must be zeros)	Reserved
368	32-bit integer	conversation_security_required (XC_NO, XC_YES)
372	32-bit integer	sync_level (CM_NONE, CM_CONFIRM, XC_EITHER_SYNC_LEVEL)
376	32-bit integer	TP_operation (XC_QUEUED_OPERATOR_STARTED, XC_QUEUED_AM_STARTED, XC_NONQUEUED_AM_STARTED, XC_QUEUED_OPERATOR_PRELOADED)
380	32-bit integer	program_type (XC_BACKGROUND, XC_FULLSCREEN, XC_PRESENTATION_MANAGER, XC_VIO_WINDOWABLE)
384	32-bit integer	Incoming_allocate_queue_limit
388	32-bit integer	Incoming_allocate_timeout
392	32-bit integer	accept_timeout
396	12-byte character string (must be zeros)	Reserved

TP_definition_length *(input)*
Specifies the length of the entry structure. Set this length to 408 (the only defined value).

return_code

Specifies the result of the call execution. The return_code can be one of the following (more details can be found in related return codes for the Define_TP verb in *Communications Server Version 4.0 System Management Programming Reference*.)

- CM_OK

  Successful completion
- CM_PROGRAM_PARAMETER_CHECK

  A parameter contains either a null or an inaccessible address.
- CM_PRODUCT_SPECIFIC_ERROR

  This return code can occur for reasons such as: out of memory, DOS call failure, etc..
- XC_PARM1_CHECK

  The *key* variable contains a value that does not match the master or service key and the Communications Manager keylock feature has been secured.
- XC_PARM3_CHECK

  The length given for the TP_definition structure is not valid.
- XC_TP_ALREADY_ACTIVE

  The TP is already defined and it is currently active or being started.
- XC_COMMUNICATION_SUBSYSTEM_ABENDED
- XC_COMMUNICATION_SUBSYSTEM_NOT_LOADED
- XC_STACK_TOO_SMALL
- XC_UNEXPECTED_DOS_ERROR
- XC_INCONSISTENT_TP_OPERATION
- XC_INVALID_CONV_SECURITY_RQD
- XC_INVALID_CONVERSATION_TYPE
- XC_INVALID_SEND_RCV_MODE
- XC_INVALID_INCOM_ALLOC_TIMEOUT
- XC_INVALID_PROGRAM_TYPE
- XC_INVALID_INCOM_ALLOC_Q_LIMIT
- XC_INVALID_ACCEPT_TIMEOUT
- XC_INVALID_SYNC_LEVEL
- XC_INVALID_TP_NAME
- XC_INVALID_TP_NAME_TYPE
- XC_INVALID_TP_OPERATION

State Changes

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

Usage Notes

If a return_code other than CM_OK is returned on this call, the TP definition is not created or changed.

Delete_TP (XCDELTP)

This call is only available in Communications Server, and only at the 32-bit C Language interface and through REXX.

Any Accept_Conversation (CMACCP) or Accept_Incoming (CMACCI) calls queued for the deleted TP are returned with the CM_PRODUCT_SPECIFIC_ERROR return code.

For nonqueued TPs, the TP definition is deleted immediately. For queued TPs, the TP definition is deleted when the active instance terminates.

More information concerning XCDEFTP and XCDELTP can be found in the Define_TP and Delete_TP verb sections of the *Communications Server Version 4.0 System Management Programming Reference*.

Format

```
CALL XCDELTP(key,
             TP_name,
             TP_name_length,
             TP_name_type,
             return_code)
```

Parameters

key *(input)*
Specifies either the master or service key, when the Communications Server keylock feature has been secured. See the *Communications Manager Installation and Configuration Guide* for details of the keylock feature.

This is an 8 byte character field. If the key is fewer than 8 characters, it must be left-justified and padded on the right with space characters.

TP_name *(input)*
Specifies the name (ASCII) of the transaction program as specified in the Define_TP call (or DEFINE_TP APPC verb).

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

TP_name_type *(input)*
Either XC_APPLICATION_TP or XC_SNA_SERVICE_TP

return_code
Specifies the result of the call execution. The return_code can be one of the following:

- CM_OK

  Successful completion

- CM_PROGRAM_PARAMETER_CHECK

  A parameter contains either a null or an inaccessible address.

- CM_PRODUCT_SPECIFIC_ERROR

  This return code can occur for reasons such as: out of memory, DOS call failure, etc..

An CMACCI call queued for a deleted TP name will fail with this return code, even if the program had associated itself with other TP names that are still defined.

- XC_PARM1_CHECK

  The *key* variable contains a value that does not match the master or service key and the Communications Manager keylock feature has been secured.

- XC_PARM2_CHECK

  The *TP_name* variable contains an invalid value.

- XC_PARM3_CHECK

  The *TP_name_length* variable contains an invalid value.

- XC_PARM4_CHECK

  The *TP_name_type* variable contains an invalid value.

- XC_TP_NAME_NOT_RECOGNIZED
- XC_COMMUNICATION_SUBSYSTEM_ABENDED
- XC_COMMUNICATION_SUBSYSTEM_NOT_LOADED
- XC_STACK_TOO_SMALL
- XC_UNEXPECTED_DOS_ERROR

State Changes

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

Usage Notes

If a return_code other than CM_OK is returned on this call, the TP definition (if it already exists) is not deleted.

Register_Memory_Object (XCRMO)

This call is only available in Communications Server, and only at the 32-bit C Language interface and through REXX.

An OS/2 memory object used for send (CMSEND, CMSNDX) or receive (CMRCV, CMRCVX) buffers is not released by Communications Manager (OS/2 call DosFreeMem) until the program ends. If, however, the program uses the Register_Memory_Object (XCRMO) call to register the object with Communications Manager ***before*** using the memory object on a send or receive call, then the program can request an early release of the memory object via the Unregister_Memory_Object (XCURMO) call.

Format

```
CALL XCRMO(memobject_base_address,
           return_code)
```

Parameters

memobject_base_address *(input)*
: The base address of the memory object. Prior to using this call, the program must obtain a memory object using the DosAllocSharedMem call to OS/2. The memory object must be shareable, giveable, readable, writeable, and tiled.

return_code
: Specifies the result of the call execution. The return_code can be one of the following:

- CM_OK

  Successful completion
- CM_PROGRAM_PARAMETER_CHECK

  A parameter contains either a null or an inaccessible address, or the memobject_base_address does not point to a valid memory object having the following characteristics: shareable, giveable, readable, writeable, and tiled.
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

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

Usage Notes

1. If this call is used to register an object that already is registered, then it will have no effect, and it returns a CM_OK return code.
2. For performance reasons, it is advisable to reuse memory objects for send and receive buffers as much as possible (instead of unregistering (XCURMO) them after each read or write).

Unregister_Memory_Object (XCURMO)

This call is only available in Communications Server, and only at the 32-bit C Language interface and through REXX.

This call is used to request Communications Manager to release memory objects that have been successfully registered by the Register_Memory_Object (XCRMO).

Format

```
CALL XCURMO(memobject_base_address,
            return_code)
```

Parameters

memobject_base_address *(input)*
The base address of the memory object.

return_code
Specifies the result of the call execution. The return_code can be one of the following:

- CM_OK

  Successful completion

- CM_PROGRAM_PARAMETER_CHECK

  A parameter contains either a null or inaccessible address, or the memobject_base_address does not point to a valid memory object.

- XC_MEMORY_OBJECT_IN_USE

  The memory object cannot be unregistered because the memory object is currently in use by Communications Manager. For example, this can occur if there is an outstanding CPI-C call using that memory object.
  .

- XC_MEMORY_OBJECT_NOT_REG

  The memory object is not currently registered with Communications Manager.

- CM_PRODUCT_SPECIFIC_ERROR

State Changes

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

Usage Notes

For performance reasons, it is advisable to reuse memory objects for send and receive buffers as much as possible (instead of unregistering (XCURMO) them after each read or write).

OS/2 Extension Calls—Conversation

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

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

Table 35 lists the Communications Manager conversation extension calls and briefly describes their functions. For the calls that set the conversation security characteristics (used with the Allocate (CMALLC) call), the program also can obtain the current values of these characteristics, except for *security_password*. This characteristic can be set, but it cannot be extracted; this restriction is intended to reduce the risk of unintentional or unauthorized access to passwords.

Table 35. List of Communications Manager Conversation Calls

Call	Pseudonym	Description
XCECST	Extract_Conversation_Security_Type	Returns the current value of the *conversation_security_type* characteristic.
XCECSU	Extract_Conversation_Security_User_ID	Returns the current value of the *security_user_ID* characteristic.
XCINCT	Initialize_Conv_For_TP	Initializes a new conversation for the specified TP.
XCSCSP	Set_Conversation_Security_Password	Sets the value of the *security_password* characteristic.
XCSCST	Set_Conversation_Security_Type	Sets the value of the *conversation_security_type* characteristic.
XCSCSU	Set_Conversation_Security_User_ID	Sets the value of the *security_user_ID* characteristic.

Extract_Conversation_Security_Type (XCECST)

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

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

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

- XC_SECURITY_NONE
- XC_SECURITY_SAME
- XC_SECURITY_PROGRAM

See "Set_Conversation_Security_Type (XCSCST)" on page 616 for a description of these values.

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

- CM_OK
  Successful completion.
- CM_PROGRAM_PARAMETER_CHECK
  This return code indicates one of the following:
  – The *conversation_ID* specifies an unassigned conversation identifier.
  – The address of one of the variables is not valid.

State Changes

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

Usage Notes

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

Extract_Conversation_Security_User_ID (XCECSU)

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

The XCECSU extension call was in Communications Manager prior to the time the Extract_Security_User_ID (CMESUI) call was part of the CPI-C architecture. For program migration purposes, the XCECSU call continues to be supported by Communications Manager.

The Extract_Security_User_ID (CMESUI) call is only available in Communications Server, and only at the 32-bit C Language interface and through REXX.

The XCECSU call provides the same function as the CMESUI call. However, it has the following differences in allowable parameters when used in releases prior to Communications Server:

1. *security_user_ID_length* can be a maximum of 8.
2. *conversation_security_type* can be set to one of the following parameters:
   - XC_SECURITY_NONE
   - XC_SECURITY_SAME
   - XC_SECURITY_PROGRAM

Initialize_Conv_For_TP (XCINCT)

A program uses the Initialize_Conv_For_TP (XCINCT) call to initialize values for various conversation characteristics before the conversation is allocated (with a call to Allocate).

XCINCT processing is similar to CMINIT processing described in "Initialize_Conversation (CMINIT)" on page 628. In addition, the XCINCT call allows the conversation being initialized to be associated with a specific TP instance.

Format

```
CALL XCINCT(conversation_ID,
            sym_dest_name,
            CPIC_TP_ID,
            return_code)
```

Parameters

conversation_ID *(output)*
Specifies the conversation identifier.

sym_dest_name *(input)*
Specifies the symbolic destination name.

CPIC_TP_ID *(input)*
Specifies the TP instance as identified by its CPIC TP ID.

If the CPIC_TP_ID is specified, the conversation being initialized is associated with that TP instance.

If the CPIC_TP_ID is set to zeros, the following rules apply:

- 0 active TP instances

  If there are no active TP instances for this OS/2 process (if no prior CMACCP, CMINIT, or XCSTP call has completed successfully), the program creates a new TP instance and initializes a new conversation.

- 1 active TP instance

  If there is one active TP instance for this OS/2 process, the program initializes a new conversation and associates it with that active TP instance.

- More than 1 active TP instance

  If there is more than one active TP instance for this OS/2 process, a return code of CM_PRODUCT_SPECIFIC_ERROR is returned, and the program creates an error log entry.

return_code *(output)*

Specifies the result of the call execution. The *return_code* can be one of the following:

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

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

Usage Notes

See Table 38 on page 628 and notes 1 through 4 on page 629.

Set_Conversation_Security_Password (XCSCSP)

A program issues the Set_Conversation_Security_Password (XCSCSP) call to set the access security password for a conversation.

The XCSCSP extension call was in Communications Manager prior to the time the Set_Conversation_Security_Password (CMSCSP) call was part of the CPI-C architecture. For program migration purposes, the XCSCSP call continues to be supported by Communications Manager.

The Set_Conversation_Security_Password (CMSCSP) call is only available in Communications Server, and only at the 32-bit C Language interface and through REXX.

The XCSCSP call provides the same function as the CMSCSP call. However, it has the following difference in allowable parameters when used in releases prior to Communications Server: *security_password._length* can be a maximum of 8.

Set_Conversation_Security_Type (XCSCST)

A program issues the Set_Conversation_Security_Type (XCSCST) call to set the security type for the conversation.

The XCSCST extension call was in Communications Manager prior to the time the Set_Conversation_Security_Type (CMSCST) call was part of the CPI-C architecture. For program migration purposes, the XCSCST call continues to be supported by Communications Manager.

The Set_Conversation_Security_Type (CMSCST) call is only available in Communications Server, and only at the 32-bit C Language interface and through REXX.

The XCSCST call provides the same function as the CMSCST call. However, it has the following differences in allowable parameters when used in releases prior to Communications Server:

conversation_security_type can be set to one of the following parameters:

- XC_SECURITY_NONE
- XC_SECURITY_SAME
- XC_SECURITY_PROGRAM

Set_Conversation_Security_User_ID (XCSCSU)

A program issues the Set_Conversation_Security_User_ID (XCSCSU) call to set the access security user ID for a conversation.

The XCSCSU extension call was in Communications Manager prior to the time the Set_Conversation_Security_User_ID (CMSCSU) call was part of the CPI-C architecture. For program migration purposes, the XCSCSU call continues to be supported by Communications Manager.

The Set_Conversation_Security_User_ID (CMSCSU) call is only available in Communications Server, and only at the 32-bit C Language interface and through REXX.

The XCSCSU call provides the same function as the CMSCSU call. However, it has the following differences in allowable parameters when used in releases prior to Communications Server: *security_user_ID_length* can be a maximum of 8.

OS/2 Extension Calls—Transaction Program Control

Transaction program control calls permit a program to end or start a TP instance, or to determine the CPIC TP ID of a TP instance. See "TP Instances for Communications Manager" on page 625.

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

Table 36 lists the transaction program control call names and gives a brief description of their function.

Table 36. List of Communications Manager Transaction Program Control Calls

Call	Pseudonym	Description
XCENDT	End_TP	Ends the specified TP instance.
XCETI	Extract_TP_ID	Returns the CPIC TP ID for the specified *conversation_ID*.
XCSTP	Start_TP	Starts a new TP instance.

End_TP (XCENDT)

A program uses the End_TP (XCENDT) call to request that CPI Communications release any resources held by CPI Communications for an active TP instance, including resources held for all conversations associated with the TP instance. This call allows a reusable resource to be used consecutively among many TP instances instead of locking the resource indefinitely in CPI Communications.

When processing End_TP, Communications Manager issues the APPC TP_ENDED verb for the specified TP instance. Upon completion of the TP_ENDED verb, Communications Manager releases the control blocks associated with that TP instance.

Format

```
CALL XCENDT (CPIC TP ID,
             type,
             return_code)
```

Parameters

CPIC TP ID *(input)*

Specifies the TP instance as identified by its CPIC TP ID.

When the CPIC TP ID is set to zeros, the following rules apply:

- 0 active TP instances

  If there are no active TP instances for this OS/2 process (if no prior CMACCP, CMINIT, or XCSTP call has completed successfully), a return code of CM_PROGRAM_STATE_CHECK is returned, and Communications Manager creates an error log entry.

- 1 active TP instance

  If there is one active TP instance for this OS/2 process, it is ended.

- More than 1 active TP instance

  If there is more than one active TP instance for this OS/2 process, a return code of CM_PRODUCT_SPECIFIC_ERROR is returned, and Communications Manager creates an error log entry.

type *(input)*

Specifies how resources held for a TP instance will be released. The *type* can be one of the following:

- XC_SOFT
  Specifies that the TP instance will wait for all active CPI Communications calls to complete.

- XC_HARD
  Specifies that all active CPI Communications calls for this TP instance are overridden and termination completes. It also ends the sessions being used by the conversations of that TP instance. Both sides of the conversation may receive conversation failure return codes. XC_HARD is not intended for the typical program, but for more complex CPI Communications applications.

Note: Prior to Communications Server, only one XCENDT call with a type of XC_HARD can be issued for a CPI Communications TP instance. If a second XC_HARD call is issued, a return code of CM_PRODUCT_SPECIFIC_ERROR is returned and Communications Manager creates an error log entry.

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

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
- CM_PRODUCT_SPECIFIC_ERROR
- CM_PROGRAM_STATE_CHECK

This value indicates CPIC TP ID is set to zeros, and there are no active TP instances.

State Changes

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

Usage Notes

None.

Extract_TP_ID (XCETI)

A program uses the Extract_TP_ID (XCETI) call to obtain the CPIC TP ID for a specified conversation.

Format

```
CALL XCETI (conversation_ID,
            CPIC TP ID,
            return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

CPIC TP ID *(output)*
Specifies the CPIC TP ID.

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

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

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

Usage Notes

None.

Start_TP (XCSTP)

A program uses the Start_TP (XCSTP) call to indicate to CPI Communications that a new TP instance is to be started.

Format

```
CALL XCSTP (local_LU_alias,
            local_LU_alias_length,
            TP_name,
            TP_name_length,
            CPIC_TP_ID,
            return_code)
```

Parameters

local_LU_alias *(input)*
CPI Communications chooses the local LU alias name by the first condition that is encountered:

1. If *local_LU_alias_length* is not 0, the *local_LU_alias* value is used.
2. If the APPCLLU environment variable exists, it is used.
3. The default LU configured for this node is used.

local_lu_alias_length *(input)*
Specifies the length of the local LU alias name.

TP_name *(input)*
CPI Communications chooses the local TP name by the first condition that is encountered:

1. If *TP_name_length* is not 0, the *TP_name* value is used.
2. If the APPCTPN environment variable exists, it is used.
3. CPIC_DEFAULT_TPNAME is used.

TP_name_length *(input)*
Specifies the length of the local TP name.

CPIC_TP_ID *(output)*
Specifies the CPIC_TP_ID.

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

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

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

Usage Notes

The Start_TP (XCSTP) call is used to start a new TP instance. The program must use the End_TP (XCENDT) call to request that CPI Communications release resources held for that active TP instance, if it is desireable to reduce resource usage before the program ends. The Deallocate (CMDEAL) call releases resources for the specified conversation, but not for the TP instance.

OS/2 Special Notes

The following sections contain information that OS/2 programmers should consider when writing programs that issue CPI Communications calls:

Migration to Communications Server

When migrating from prior releases of Communications Manager to Communications Server some situations that resulted in a CM_PRODUCT_SPECIFIC_ERROR(TP_busy) return code will now complete with the return code CM_OPERATION_NOT_ACCEPTED. Refer to “TP Instances for Communications Manager” on page 625.

Multi-threaded CPI-C Programs

This section contains information that OS/2 programmers should consider when writing multi-threaded programs that issue CPI-C calls to Communications Manager as follows:

- Scope of CPI-C conversation_ID and CPIC TP ID
  All CPI-C calls issued within one OS/2 process are considered by Communications Manager to be part of ***one*** program. The scope of CPI-C information (such as conversation_ID or CPIC TP ID) kept for a program is the OS/2 process in which it is executing. For example, if a program issues an Accept_Conversation (CMACCP) call on thread 1, the conversation_ID that is obtained can be used by that same program on thread 2 (of the same OS/2 process) to issue the Send_Data call.
- Specifying TP name
  CMSLTP and CMRLTP operate on only one list of TP names that is unique per program, regardless of which thread (in that OS/2 process) issues the CMSLTP or CMRLTP call. Therefore, multi-threaded CPI-C programs should coordinate the use of these calls.

  Note: when the Accept_Incoming (CMACCI) call is being used, and only one TP name needs to be specified, an alternative method is to use the Set_TP_Name (CMSTPN). Refer to “Accept_Incoming (CMACCI)” on page 450. This takes effect per conversation, and removes the need to coordinate among other threads.
- OS/2 environment variables
  When a multi-threaded operator-started program is using OS/2 environment variable(s), such as `APPCTPN` or `APPCLLU` (for example for the CMACCP, CMACCI, CMINIT, XCINCT calls), setting of this variable by the program should be serialized across threads (if using a different value of the variable per thread).
- CMWCMP conficting OOIDs
  Refer to “Wait_For_Completion (CMWCMP)” on page 631.

Considerations for CPI Communications Calls

This section describes TP instances, followed by CPI Communications calls that require special consideration when one is writing a CPI Communications program to be run on a Communications Manager system. Each call needing special attention is discussed in alphabetical order by call name.

Refer to Table 15 on page 109 to determine what releases of Communications Manager support which architected CPI-C calls.

Note: Explanations of error return codes whose causes on Communications Manager differ from those defined for CPI Communications are not included in this section. See "Diagnosing Errors" on page 448 for this information.

TP Instances for Communications Manager

Within an OS/2 process, CPI Communications creates an executable instance of a transaction program (TP instance) when a program issues any of the following CPI Communications calls:

- Accept_Conversation (CMACCP) call

  Note: Multiple CMACCP calls can be issued within an OS/2 process. Each CMACCP creates a new TP instance.

- Accept_Incoming (CMACCI) call

  Note: Multiple CMACCI calls can be issued within an OS/2 process. Each CMACCI creates a new TP instance.

- Initialize_Conversation (CMINIT) call, if all of the following conditions are met (within this OS/2 process):
  - No prior CMINIT call has been issued
  - No prior CMACCP calls have been issued
  - No prior XCSTP calls have been issued

- Start_TP (XCSTP) call

  Note: Multiple XCSTP calls can be issued within an OS/2 process. Each XCSTP creates a new TP instance.

CPI Communications represents the TP instance by using a transaction program identifier, or CPIC TP ID. CPI Communications converts each CPI Communications call, other than Extract and Set calls, to an APPC verb and makes a call across its APPC interface to process the verb. Each CPIC TP ID is uniquely associated with an APPC TP identifier (TP ID). Each APPC verb includes the TP ID as a parameter.

CPI Communications associates with each TP instance the logical unit of work identifier and access security information (if any) that it obtains when it starts the TP instance. It maintains the correlation of this information to the CPIC TP ID until the TP instance ends.

Each TP instance remains active until the program ends, or until it is explicitly ended by using the End_TP (XCENDT) call.

Each conversation is associated with only one TP instance. However, a TP instance can be associated with more than one conversation. When the TP instance ends, all associated conversations are ended.

Usage Note 1: The Initialize_Conversation (CMINIT) call is used when a program initializes a conversation within an OS/2 process that contains up to one TP instance. When initializing a conversation within an OS/2 process that contains more than one TP instance, the program must specify a particular TP instance (CPIC TP ID) and the conversation will be associated with it. This association is done by using the Initialize_Conv_For_TP (XCINCT) call.

Usage Note 2: This note is applicable to releases of Communications Manager prior to Communications Server:

For each conversation, Communications Manager CPI-C only allows one CPI-C call to be in-progress at a time. It rejects a second call for that conversation with CM_PRODUCT_SPECIFIC_ERROR(TP_busy). Also, for each TP instance, Communications Manager CPI-C only allows one total CPI-C call to be in-progress at a time for all the conversations running under that TP instance. It rejects a second call with CM_PRODUCT_SPECIFIC_ERROR(TP_busy). To avoid this situation, if a program issues calls to more than one conversation at a time, those conversations should be created under separate TP instances. For accepting conversations this is automatically done by the CMACCP call. For initializing conversations, this can be done by not using the CMINIT call, but instead using the XCSTP call followed by the XCINCT call (specifying the CPIC_TP_ID returned on the XCSTP call).

Accept_Conversation (CMACCP) or Accept_Incoming (CMACCI)

When the Accept_Conversation or Accept_Incoming call completes successfully, the following conversation characteristics are initialized:

Table 37. Additional Communications Manager Characteristics Initialized following CMACCP or CMACCI

Conversation Characteristic	Initialized Value
conversation_security_type	CM_SECURITY_SAME (or eqivalently XC_SECURITY_SAME) **Note:** This value is set regardless of the level of access security information (if any) on the inbound allocation request.
security_user_ID	The value received on the conversation startup request (10 characters in Communications Server, truncated to 8 characters in Communications Manager releases prior to Communications Server). If the conversation startup request contained no access security information, this characteristic is set to a single-space character.
security_user_ID_length	The length of *security_user_ID*.
security_password	A single-space character.
security_password_length	Set to 1.

For the Accept_Conversation (CMACCP) call, Communications Manager processes the following rules, sequentially, the first rule that is true is used to determine the TP name(s) to use for accepting an incoming conversation:

1. The TP name(s) from successfully completed Specify_Local_TP_Name (CMSLTP) call(s) for this progam. Refer to "Specify_Local_TP_Name (CMSLTP)" on page 451 for allowable TP names on the CMSLTP call.
2. A TP name set in the APPCTPN OS/2 environment variable. The environment variable must be set to a valid TP name.

Partially specified TP names and '*' (allowed on the CMSLTP call) are not allowable TP names for the APPCTPN environment variable.

If none of the above rules yields a TP name, then CMACCP completes with a CM_PROGRAM_STATE_CHECK return code.

For the rules determining the TP name used on the CMACCI call, refer to “Accept_Incoming (CMACCI)” on page 450

When an inbound allocation request arrives with this TP name specified, Communications Manager completes the call.

Notes:

1. When the APPCTPN environment variable is used to specify a TP name, the TP name set in the environment variable is case sensitive; that is, lowercase letters are not converted to uppercase. The TP name set in the environment variable is made up of ASCII characters; therefore, it must be an application TP name, not an SNA service TP name. See “Unsupported TP Names” on page 452 for an explanation of this restriction.
2. For an attach manager-started program, Communications Manager sets the APPCTPN environment variable with the TP name from the inbound allocation request for the conversation when it starts the program. It then uses the TP name from the environment variable to complete the subsequent Accept_Conversation or Accept_Incoming call from the program. Therefore, for Communications Manager to match the Accept_Conversation or Accept_Incoming call with the inbound conversation, the attach manager-started program should not set the environment variable to a different TP name.
3. Communications Manager recognizes certain error conditions while accepting conversations that are specific to its use of OS/2 environment variables. See “Diagnosing Errors” on page 448 for more details.
4. Each Accept_Conversation (CMACCP) or Accept_Incoming (CMACCI) call starts a new TP instance. The program must use the End_TP (XCENDT) call to request that CPI Communications release resources held for that active TP instance, if it is desireable to reduce resource usage before the program ends. The Deallocate (CMDEAL) call releases resources for the specified conversation, but not for the TP instance.

Extract_Conversation_Context (CMECTX)

The Extract_Conversation_Context (CMECTX) call currently returns a context that is the left justified 12-byte CPIC TP ID padded by zeros on the right, to 32-bytes total length.

Since contexts are subject to future architectural changes, the content of this field is subject to change. In addition, dependencies on the context are not portable across platforms.

Extract_Secondary_Information (CMESI)

After a call fails that causes CPI-C to generate secondary information, the next CPI-C call for that conversation should be the Extract_Secondary_Information (CMESI) call (because other calls can reset the secondary information).

If a CMACCP or CMACCI call is not successful, CPI-C will return a temporary conversation ID for use on the CMESI call. It is recommended that the CPI-C program issue the CMESI as soon as possible, since the CPI-C implementation will eventually delete the secondary information and conversation ID.

Communications Server provides limited support for the CMESI call as follows:

- The only supported values of the call_ID parameter for a CMESI call are the following:
  - CM_CMACCI
  - CM_CMACCP
  - CM_CMALLC
  - CM_CMCFM
  - CM_CMDEAL
  - CM_CMPTR
  - CM_CMRCV
  - CM_CMSEND
  - CM_CMSERR

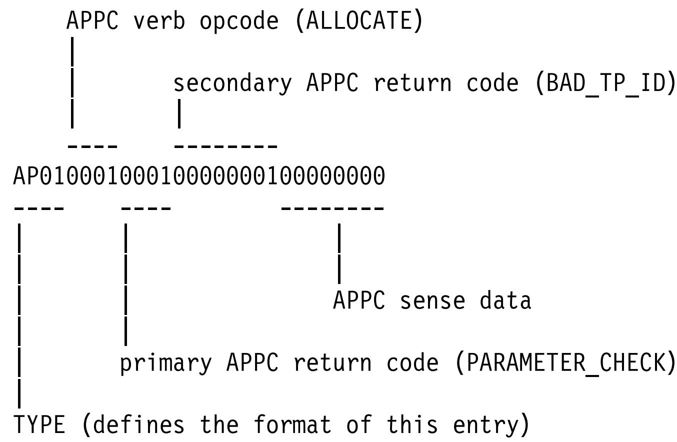
  For all other values of this parameter, the call is returned with the return code CM_NO_SECONDARY_INFORMATION.
- CMESI returns additional secondary information if the previous CMACCP, CMACCI, or CMALLC call was rejected with a CM_PRODUCT_SPECIFIC_ERROR return code and error information is available to Communications Manager (for example, a failure in a call to the APPC component of Communications Manager). For errors detected by OS/2 rather than Communications Manager, this call returns CM_NO_SECONDARY_INFORMATION if Communications Manager has no additional information about the failure.
- CMESI returns additional secondary information for CRM-specific associated return codes.

The layout of the “Additional information from the implementation,” which is a subfield of secondary information, is described in the Communications Manager C Language header file (cpic.h). The format of the field is determined by what type of error has occurred.

The following is an example of a secondary information buffer when the call failed due to an error detected within the APPC component of Communications Manager.

```
4003;;CMR0323: Common Programming Interface Communications (CPI-C) received
an unexpected return code from advanced program-to-program communications
(APPC).;;;;;;AP0100010001000000010000000
```

The following is an explanation of the “Additional information from the implementation” field included in the above example. Refer to the *Communications Manager/2 APPC Programming Reference* for the meaning of the APPC fields.



Initialize_Conversation (CMINIT)

When Initialize_Conversation completes successfully, the Communications Manager specific characteristics are initialized as shown in Table 38 on page 628.

Table 38. Additional Communications Manager Characteristics Initialized following CMINIT

Conversation Characteristic	Initialized Value
conversation_security_type	Security type from side information referenced by *sym_dest_name*. If a blank *sym_dest_name* is specified, *conversation_security_type* is set to CM_SECURITY_SAME (or equivalently XC_SECURITY_SAME).
security_user_ID	User ID from side information referenced by *sym_dest_name*, if *conversation_security_type* is CM_SECURITY_PROGRAM (or equivalently XC_SECURITY_PROGRAM); otherwise, a single space character. If a blank *sym_dest_name* is specified, *security_user_ID* is set to a single space character.
security_user_ID_length	Length of security user ID, if *conversation_security_type* is CM_SECURITY_PROGRAM (or equivalently XC_SECURITY_PROGRAM); otherwise, 1. If a blank *sym_dest_name* is specified, *security_user_ID_length* is set to 1.
security_password	Password from side information referenced by *sym_dest_name* if *conversation_security_type* is CM_SECURITY_PROGRAM (or equivalently XC_SECURITY_PROGRAM); otherwise, a single space character. If a blank *sym_dest_name* is specified, *security_password* is set to a single space character.
security_password_length	Length of security password, if *conversation_security_type* is CM_SECURITY_PROGRAM (or equivalently XC_SECURITY_PROGRAM); otherwise, 1. If a blank *sym_dest_name* is specified, *security_password_length* is set to 1.

Initialize_Conversation Notes:

1. If an operator-started CPI Communications program is to be run on a local LU other than the default LU configured for the node, either the operator or the program must set the local LU name in an OS/2 environment variable named `APPCLLU` before the program issues the CMINIT call.
2. The `APPCLLU` environment variable is used only for an operator-started CPI Communications program that issues the Initialize_Conversation call to start the TP instance. The LU name set in the environment variable is case sensitive; that is, lowercase letters are not converted to uppercase.
3. The `APPCTPN` environment variable is used to obtain the local TP name for any type CPI Communications program (operator-started or attach manager-started) that issues the Initialize_Conversation call to start the TP instance. Communications Manager sets the environment variable to a default local TP name of `CPIC_DEFAULT_TPNAME`. The operator or program may set the `APPCTPN` environment variable to a different TP name before the program issues the

Initialize_Conversation call, if a different local TP name is desired. Communications Manager does not send this local TP name outside the node.

4. Communications Manager recognizes certain error conditions on the Initialize_Conversation call that are specific to its use of OS/2 environment variables. See “Diagnosing Errors” on page 448 for more details.

5. For CMINIT, the following rules apply:

   - 0 active TP instances

     If there are no active TP instances for this OS/2 process (if no prior CMACCP, CMINIT, or XCSTP has completed successfully), Communications Manager creates a new TP instance and initializes a new conversation.

   - 1 active TP instance

     If there is one active TP instance for this OS/2 process, Communications Manager initializes a new conversation and associates it with that active TP instance.

   - More than 1 active TP instance

     If there is more than one active TP instance for this OS/2 process, a return code of CM_PRODUCT_SPECIFIC_ERROR is returned and Communications Manager creates an error log entry.

     When a CMINIT call starts a TP instance, the TP instance remains active until the End_TP (XCENDT) is issued or the program ends.

     For multiple TP instances, use an XCINCT call to initialize a new conversation.

Receive (CMRCV)

When the Receive call is receiving data from a basic conversation, the 2-byte logical record length, or LL, field of the data is in System/370 format, with the left byte being the most significant. Depending on the programming language used, the program might have to reverse the bytes to use the field value in an integer operation.

Communications Manager does not perform any EBCDIC-to-ASCII translation on the data before placing it in the *buffer* variable.

See “Performance Considerations For Using Send/Receive Buffers” on page 500 for additional information concerning data buffers.

Send_Data (CMSEND)

When the Send_Data call sends data on a basic conversation, the 2-byte logical record length, or LL, field of the data must be in System/370 format, with the left byte being the most significant. If the program obtains this value from an integer operation, it might have to reverse the bytes before issuing the call, depending on the programming language used.

Communications Manager does not perform any ASCII-to-EBCDIC translation on the data when it sends the data from the *buffer* variable.

See “Performance Considerations For Using Send/Receive Buffers” on page 500 for additional information concerning data buffers.

Send_Expedited_Data (CMSNDX)

A CMSNDX call will not complete if the partner program has not yet received the data from a prior CMSNDX call for that same conversation. This is illustrated with the following example where sequential non-blocking Send_Expedited_Data (CMSNDX) calls are issued to Communications Manager:

1. A program issues a CMSNDX call, with a resulting CM_OK return code.
2. A second CMSNDX call is issued immediately for that same conversation.

   If the partner program has not yet issued the Receive_Expedited_Data (CMRCVX) call to obtain the data from the first CMSNDX call, then the result of the second CMSNDX call is a CM_OPERATION_INCOMPLETE return code. The second CMSNDX call will not complete until the partner has issued the Receive_Expedited_Data call to receive the data for the first CMSNDX call.

Set_Partner_LU_Name (CMSPLN)

The program can set the *partner_LU_name* characteristic to either an alias or a network name. Alias and network names are distinguished from each other on this call as follows:

- An alias name is 1–8 characters and does not contain a period. Communications Manager retains alias names in ASCII, without translating them to EBCDIC.
- A network name is 2–17 characters, with a period separating the network ID (0–8 characters) from the network LU name (1–8 characters). If the network name does not include a network ID, the period must still be inserted preceding the network LU name, to distinguish the name as a network name instead of an alias name.

Set_Sync_Level (CMSSL)

See “Set_Sync_Level (CMSSL)” on page 451 for information on OS/2 support of synchronization levels.

Set_Queue_Processing_Mode (CMSQPM)

For the Set_Queue_Processing_Mode (CMSQPM) call, a valid memory area must be allocated for the user_field parameter, even if this parameter is not being used by the program.

Test_Request_To_Send (CMTRTS)

The CMTRTS call does not support CM_EXPEDITED_DATA_AVAILABLE.

Wait_For_Completion (CMWCMP)

For the Wait_For_Completion (CMWCMP) call a valid memory area must be allocated for the user_field_list parameter, even if this parameter is not being used by the program.

For CMWCMP, 512 is the maximum OOID_list_count that can be specified.

If on a CMWCMP call, a valid OOID (for example, OOIDx) is specified but there is no outstanding operation for that OOID (at the time the CMWCMP call is issued by the program), then the following will occur:

- If there is at least one OOID on the list for which there is an outstanding operation, then OOIDx will be ignored (treated as a NULL (integer zero) OOID).

- If there are valid OOIDs on the list, but all of them have no outstanding operations, then the following is returned: CM_PROGRAM_STATE_CHECK return code.

Conflicting OOIDs:
Suppose OOIDx has outstanding operations, and it is being waited on by a CMWCMP call (denoted CMWCMP1). Before CMWCMP1 completes, if another call (CMWCMP2) waits on OOIDx, then OOIDx will be ignored (treated as a NULL (integer zero) OOID) with respect to CMWCMP2. Moreover, OOIDx will not be waited on by CMWCMP2, even if CMWCMP1 completed for a reason other than OOIDx. To avoid this situation, do one of the following:

- Do not use the same OOID simultaneously on more than one CMWCMP call.
- Use the timeout paramater on CMWCMP to allow your program to get control to periodically reissue the CMWCMP call.

Characteristics, Fields, and Variables

This section defines the values and data types for the additional characteristics, fields, and variables used with the Communications Manager calls. It also includes the CPI Communications variables for which Communications Manager imposes certain restrictions.

The following distinctions are made regarding characteristics, fields, and variables:

Characteristic
: An internal parameter of a given conversation whose value is maintained within the CPI Communications component. The value of a conversation characteristic is initialized during the Initialize_Conversation (CMINIT) or Accept_Conversation (CMACCP) call for that conversation. The value may be changed subsequently using a Set call.

Field
: An element of a data structure. The data structure itself is specified as a variable on the Set_CPIC_Side_Information and Extract_CPIC_Side_Information calls. A field can supply a value as input on a call, or return a value as output from a call.

Variable
: A parameter specified on a call. A variable can supply a value as input on a call, or return a value as output from a call.

Note: Communications Manager does not support all values of the conversation characteristics and variables described in Appendix A, “Variables and Characteristics.”

Communications Manager Native Encoding

The native encoding for specifying certain character string variables and fields on Communications Manager is ASCII. These variables and fields are:

- *key*
- *mode_name**
- *partner_LU_name* (as an alias name)
- *partner_LU_name* (as a network name)*
- *security_password**
- *security_user_ID**
- *sym_dest_name*

- *TP name* (as an application TP name)*
- *TP name* (as an SNA service TP name, excluding first character)*

The variables and fields indicated by an asterisk (*) are translated from ASCII to EBCDIC on input, and from EBCDIC to ASCII on output. The others are retained in ASCII.

Communications Manager performs translation between the ASCII characters having code points in the range X'20' through X'7E' and the corresponding EBCDIC characters. The ASCII characters are those defined in the ASCII code page 850. However, this range of characters is common across all ASCII code pages. The EBCDIC characters are those defined in the EBCDIC code page 500.

Communications Manager translates all of the characters from character set 00640 (including the space character) and the 13 additional characters listed in Table 39. Characters outside character set 00640 and not shown in this table are not translated and remain unchanged.

Table 39. Additional Communications Manager Characters Translated between ASCII and EBCDIC

Graphic	Description
[	Left bracket
!	Exclamation point
]	Right bracket
$	Dollar sign
^	Caret
`	Right prime
#	Number sign
@	At sign
~	Tilde
|	Vertical bar
{	Left brace
}	Right brace
\	Back slash

Variable Types and Lengths

Table 40 defines the data type, character set usage, and length of fields and variables used for OS/2 extension calls. It also includes CPI Communications variables for which Communications Manager imposes certain restrictions.

The variables Communications Manager uses for its conversation calls are the same two types of variables that CPI Communications uses: integer and character. The variables used by Communications Manager for system management calls include a third type: a data structure. The data structure has both integer and character string fields.

With one exception, the program specifies the character string fields and variables using ASCII characters on the Set calls, and Communications Manager returns them as ASCII characters on the Extract calls. The exception to this is SNA service TP name, as noted inTable 40 .

All fields of the *side_info_entry* structure are fixed length, so character string data within these fields must be specified. The data is returned left-justified and padded on the right with ASCII space characters. In general, Communications Manager does not reject the Set_CPIC_Side_Information call when a character string field of the *side_info_entry* contains all space characters, even if the minimum length defined for the data in that field is greater than 0. The *sym_dest_name* is the exception, as noted in Table 40.

The use of character set 01134 or 00640, as defined for each field or variable, is recommended for consistency with the CPI Communications definition; however, Communications Manager does not enforce this.

Table 40. Communications Manager Variable and Field Types and Lengths

Variable or Field	Data Type	Character Set	Length
conversation_security_type	Integer	Not applicable	32 bits
CPIC TP ID	Character string	Not applicable	12 bytes
key [1, 10]	Character string	01134	8 bytes
mode_name [1, 2]	Character string	01134	0-8 bytes
partner_LU_name [3, 5] (as an alias name)	Character string	01134	1-8 bytes
partner_LU_name [1, 4] (as a network name)	Character string	01134	2-17 bytes
security_password [5]	Character string	00640	1-8, 1-10 bytes[12]
security_password_length	Integer	Not applicable	32 bits
security_user_ID [5]	Character string	00640	1-8, 1-10 bytes[13]
security_user_ID_length	Integer	Not applicable	32 bits
side_info_entry [6]	Data structure	Field dependent	124 bytes
side_info_entry_length	Integer	Not applicable	32 bits
sym_dest_name [7, 11]	Character string	01134	8 bytes
TP_name [1, 5, 8] (as an application TP name)	Character string	01134	1-64 bytes
TP_name [1, 9] (as an SNA service TP name)	Character string	01134	1-4 bytes
TP_name_type	Integer	Not applicable	32 bits

Referenced Notes:

1. The national characters @, #, and $ are also allowed as part of the key, mode name, partner LU name, and TP name. (For a TP name only, lower case alphabetic characters are also acceptable.)

2. The null string (all space characters) is a valid mode name. The program should not set the *mode_name* to CPSVCMG or SNASVCMG. Although Communications Manager allows the program to specify these mode names, it rejects a program's Allocate (CMALLC) call with a *return_code* of

CM_PARAMETER_ERROR if the conversation characteristic is set to either of these mode names.

3. A period character is not allowed as part of an alias partner LU name. An alias partner LU name might contain ASCII characters in the range X'21' to X'FE'; however, use of characters drawn from character set 01134 (plus the national characters @, #, and $) is recommended.

4. The period must be present in a network partner LU name because it distinguishes the name as a network name instead of an alias name. If the partner LU name does not have a network ID, the period must be the first character in the *partner_LU_name* variable or field.

5. The space character is not allowed as part of a partner LU name, security password, security user ID, or application TP name, because it is used as the fill character in the corresponding fields of the *side_info_entry* data structure.

6. The format of the *side_info_entry* data structure is shown in the description of "Set_CPIC_Side_Information (XCMSSI)" on page 607.

7. The *sym_dest_name* can be specified as all space characters only on the Initialize_Conversation (CMINIT) call. On all other calls that include this variable or field, the name must be 1–8 characters long.

8. An application TP name is composed entirely of ASCII characters. It cannot be a double-byte TP name—one that has a leading X'0E' byte and a trailing X'0F' byte—because Communications Manager does not support double-byte TP names. Communications Manager converts all characters of an application TP name from ASCII to EBCDIC when it includes the TP name on an allocation request.

9. An SNA service TP name is composed of a leading SNA service TP identifier byte and 0–3 additional ASCII characters; the identifier byte has a value in the range X'00' to X'0D' and X'0F' to X'3F'. An SNA service TP name may be specified only with the Set_CPIC_Side_Information call; it cannot be specified on the Set_TP_Name call.

10. The *key* variable on the Delete_CPIC_Side_Information (XCMDSI) and Set_CPIC_Side_Information (XCMSSI) calls must be at least 8 bytes long. The key within the variable may be 1–8 characters long. If the key is shorter than 8 characters, it must be left-justified in the variable and padded on the right with space characters. If the variable is longer than 8 bytes, the key is taken from the first (leftmost) 8 bytes and the remaining bytes are ignored.

11. The *sym_dest_name* variable on the Delete_CPIC_Side_Information (XCMDSI) and Extract_CPIC_Side_Information (XCMESI) calls must be at least 8 bytes long. The symbolic destination name within the variable may be 1–8 characters long on these calls. If the symbolic destination name is shorter than 8 characters, it must be left-justified in the variable and padded on the right with space characters. If the variable is longer than 8 bytes, the symbolic destination name is taken from the first (leftmost) 8 bytes and the remaining bytes are ignored.

12. The length of *security_password* can be 1-10 bytes in Communications Server, and 1-8 bytes prior to Communications Server.

13. The length of *security_user_ID* can be 1-10 bytes in Communications Server, and 1-8 bytes prior to Communications Server.

Defining and Running a CPI Communications Program on Communications Manager

This section discusses the operating parameters that can be configured for a CPI Communications program and how they affect the program's operation.

Defining a CPI Communications Program to Communications Manager

A CPI Communications program can be defined to Communications Manager by configuring a transaction program definition. The TP definition includes the OS/2 file specification (the drive, path, filename, and extension for the file containing the program), a TP name, and certain operating characteristics for the program. In writing a program that is to be started by an inbound allocation request, one should define the program to Communications Manager as non-queued, and attach manager-started. This definition allows multiple instances of the program to be started concurrently when Communications Manager receives multiple inbound allocation requests for the TP name defined for the program.

The TP name must be an application TP name. A local CPI Communications program cannot be an SNA service TP; that is, the program cannot accept a conversation started by an inbound allocation request specifying an SNA service TP name. See “Unsupported TP Names” on page 452 for more information on this restriction.

A CPI Communications program that is to be started by an inbound allocation request does not require a TP definition if the program is not a REXX program; see “Using Defaults for TP Definitions” for more details. (See “Starting a REXX Program” on page 440 for an explanation of why REXX programs that are started by inbound allocation requests require a TP definition.)

If the program issues the Initialize_Conversation (CMINIT) call to start the TP instance, Communications Manager starts the TP instance on the default local LU configured for the node. If the operator (or program) wants the program to be started on a different LU, it must set the alias name for the desired local LU in the OS/2 environment variable named `APPCLLU`. Communications Manager then starts the program on that LU. A program that issues the Initialize_Conversation call to start the TP instance does not require a TP definition.

If the program issues the Accept_Conversation (CMACCP) call, Communications Manager starts the TP instance on the LU that receives the inbound allocation request.

Using Defaults for TP Definitions

Communications Manager provides defaults for attach manager-started programs. The defaults eliminate the need to explicitly configure TP definitions to Communications Manager for these programs. To use these defaults for a CPI Communications program:

- The TP name must be the same as the name of the file; Communications Manager converts the TP name from the inbound allocation request to ASCII and uses the ASCII name to start the program.
- The program must be in the OS/2 subdirectory for default TPs, or in an OS/2 subdirectory of the current path.

- The option for how default TPs are started must be Non-queued, Attach Manager started; the other choices are for queued (single-instance) programs, which do not apply to CPI Communications programs.
- The presentation type for the program must agree with the execution environment the program requires. The options are:
  - Presentation Manager
  - VIO-windowable
  - Full screen
  - Background
- The choice of Yes or No for whether default TPs require conversation security[6] must agree with what the CPI Communications program requires.

These default TP definitions apply across all Communications Manager programs—APPC programs and CPI Communications programs alike. Explicitly configure TP definitions for all CPI Communications programs having operating characteristics that differ from these default definitions.

Communications Manager Use of OS/2 Environment Variables

Communications Manager makes use of certain OS/2 environment variables when processing CPI Communications calls. The environment variables become part of the OS/2 process created when the CPI Communications program is started. These environment variables are:

APPCTPN
: During an Accept_Conversation (CMACCP) call, Communications Manager obtains the TP name from this environment variable. It completes the call when an inbound allocation request carrying the same TP name arrives, or if one is already waiting. The operator or program must set the TP name in this environment variable for an operator-started program that uses the CMACCP call. Communications Manager sets the TP name from an inbound allocation request in this environment variable when it starts an attach manager-started program.

  Refer to "Accept_Incoming (CMACCI)" on page 450 for use of the APPCTPN environment variable with the CMACCI call.

  During an Initialize_Conversation call, Communications Manager obtains the local TP name from this environment variable when the call starts the TP instance. If the operator or program does not set this environment variable, Communications Manager uses CPIC_DEFAULT_TPNAME as a default name for the local TP instance.

APPCLLU
: During an Initialize_Conversation call that starts the TP instance, Communications Manager obtains the local LU name from this environment variable. It then starts the TP instance on this local LU. If this environment variable is not set when the program issues an Initialize_Conversation call that starts the TP instance, Communications Manager starts the TP instance on the default local LU configured for the node.

---

[6] The option of Yes for conversation security required means the inbound allocation requests that start default TPs must include LU 6.2 access security information.

Stack Size

Communications Manager requires that CPI-C programs have a minimum stack size of 4500 bytes. However, programs using the 32-bit interface are required to have a minimum stack size of 8KB (8192 bytes).

Performance Considerations For Using Send/Receive Buffers

This section provides information related to performance in the Communications Manager environment. For performance reasons, it advisable for CPI-C programs to reuse data buffers.

The Extract_Maximum_Buffer_Size (CMEMBS) is available in Communications Manager/2 Version 1.11 or later. It should be used by programs to dynamically determine maximum send/receive buffer sizes supported by Communications Manager. In Communications Server, the maximum was raised from 32767 to 65535 bytes.

When a program issues a call that supplies a data buffer, such as Send_Data (CMSEND), Send_Expedited_Data (CMSNDX), Receive (CMRCV), and Receive_Expedited_Data (CMRCVX), Communications Manager determines whether the data buffer can be shared (giveable memory) across OS/2 processes. If it can, Communications Manager uses the buffer directly as the source or destination of data. If the buffer cannot be shared, Communications Manager must copy the data between memory it allocates and the program's buffer. Therefore, by using memory that can be shared for the buffer, the program can avoid the extra copying of the data and the memory allocation that Communications Manager would otherwise do on each applicable call.

It is recommended for future migration that buffers be defined with the OS/2 tiled memory allocation option.

A program can use shareable memory for its data buffer by explicitly allocating the memory, using OS/2 calls. Depending on the application, the program might have to allocate the shareable memory only once, when it is started, and use that memory for its send and receive buffers for the remainder of its execution. For performance reasons, shareable data buffers passed to CPI-C on Send_Data (CMSEND), Send_Expedited_Data (CMSNDX), Receive (CMRCV), and Receive_Expedited_Data (CMRCVX), are internally registered within Communications Manager and not freed until the CPI-C program (i.e. that OS/2 process) ends. This means that the memory will not be freed by OS/2 at the time the program frees it. However, for programs running under Communications Server, calls are available to allow the program to free memory before the CPI-C program (i.e. that OS/2 process) ends. Refer to “Register_Memory_Object (XCRMO)” on page 469 and “Unregister_Memory_Object (XCURMO)” on page 470.

For more information on memory allocation for send and receive buffers, and other performance suggestions, refer to the *APPC Programming Reference* for the specific product being used.

Exit List Processing

If a CPI Communications program registers for OS/2 exit list processing, it must register outside the range of priorities specified in the *APPC Programming Reference* for the specific product being used.

When an OS/2 process ends, APPC gets control (before a CPI Communications program gets control) during exit list processing to clean up resources Consequently, CPI Communications calls made by a program during exit list processing will not be successful.

For other exit list considerations, refer to "How Dangling Conversations Are Deallocated" on page 447 and "Ending a REXX Program" on page 442.

Sample Program Listings for OS/2

This section provides listings of some sample programs that show how Communications Manager calls are made using the SAA languages that these products support.

The listings are provided for tutorial purposes to show how the calls are made. They are not intended to show complete applications or the most efficient way of performing a function.

OS/2 C Sample Programs

This program is a C language sample program that sets CPI-C side information.

SETSIDE.C

```
/**********************************************************************/
/*   This program is a C language sample program that sets CPI-C      */
/*   side information.                                                */
/*                                                                    */
/*   Note:  Before running this program, the Communications           */
/*          Manager keylock feature must either be secured with a     */
/*          service key of "SVCKEY" or not be secured at all.         */
/**********************************************************************/

#include <os2.h>

#include <string.h>
#include <stdio.h>
#include <cmc.h>

int main (void)

{
  SIDE_INFO side_info;
  CM_INT32  side_info_length;
  CM_RETURN_CODE rc;

  side_info_length = sizeof(SIDE_INFO);
  /************************************************/
  /* Set fields in side_info structure            */
  /************************************************/
  memset(&side_info,' ',sizeof(side_info));

  memcpy(side_info.sym_dest_name,"SYMDEST1",8);
  memcpy(side_info.partner_LU_name,"ALIASLU2",8);
  side_info.TP_name_type = XC_APPLICATION_TP;
  memcpy(side_info.TP_name,"MYTPN",5);
  memcpy(side_info.mode_name,"        ",8);

  side_info.conversation_security_type = XC_SECURITY_PROGRAM;
  memcpy(side_info.security_user_ID,"MYUSERID",8);
  memcpy(side_info.security_password,"XXXXXXXX",8);

  /***********************************************************/
  /* Call XCMSSI to define the side info                     */
  /***********************************************************/
  /* Note: The key must be 8 bytes long (blank pad if needed) */
  /***********************************************************/
  xcmssi("SVCKEY  ", &side_info, &side_info_length, &rc);
  printf("return code from xcmssi is: %d \n", rc);

  return(0);
}
```

OS/2 COBOL Sample Programs

This program is an example of the function available through the CPI-C extensions provided. *

DEFSIDE.CBL

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.       DEFSIDE.
***************************************************************
* THIS PROGRAM IS AN EXAMPLE OF THE FUNCTION AVAILABLE        *
* THROUGH THE CPI-C EXTENSIONS PROVIDED.                      *
*                                                             *
* PURPOSE: DEFINE CPI-C SIDE INFORMATION AND DISPLAY RESULT   *
*                                                             *
* INPUT:   SIDE-INFORMATION STRUCTURE.                        *
*                                                             *
* OUTPUT:  CPI-C SIDE INFORMATION TABLE IS UPDATED TO         *
*          REFLECT INPUT STRUCTURE.                           *
*                                                             *
* NOTE:    FOR THIS SAMPLE PROGRAM, THE KEY FIELD (TEST-KEY), *
*          SUPPORTING THE COMMUNICATIONS MANAGER              *
*          KEYLOCK FEATURE, IS SET TO SPACES.  AS A RESULT,   *
*          THIS PROGRAM WILL RUN SUCCESSFULLY ONLY WHEN       *
*          THE KEYLOCK FEATURE IS NOT SECURED.                *
*                                                             *
***************************************************************
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. PS-2.
 OBJECT-COMPUTER. PS-2.
 SPECIAL-NAMES.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
 I-O-CONTROL.
*
 DATA DIVISION.
 FILE SECTION.
 WORKING-STORAGE SECTION.

 01  TEST-KEY                    PIC  X(1)   VALUE SPACES.
 01  TEST-ENTRY-NUMBER           PIC  9(9)   VALUE 0 COMP-4.

 01  CM-ERROR-DISPLAY-MSG        PIC  X(40)  VALUE SPACES.

**********************************************
* USE THE CPI COMMUNICATIONS PSEUDONYM FILES *
**********************************************
     COPY CMCOBOL.

 LINKAGE SECTION.

 EJECT
*
 PROCEDURE DIVISION.
******************************************************************
*************************  START OF MAINLINE  *******************
******************************************************************
 MAINLINE.

     PERFORM SIDE-INITIALIZE
        THRU SIDE-INITIALIZE-EXIT.
     PERFORM SIDE-DISPLAY
```

```
           THRU SIDE-DISPLAY-EXIT.
        PERFORM CLEANUP
           THRU CLEANUP-EXIT.
        STOP RUN.
******************************************************************
***************************  END OF MAINLINE  ********************
******************************************************************
*
 SIDE-INITIALIZE.
     INITIALIZE SIDE-INFO-ENTRY REPLACING NUMERIC BY 0
                                   ALPHABETIC BY " ".
     MOVE "CREDRPT" TO SI-SYM-DEST-NAME.
******************************************************************
* CHANGE THE SI-PARTNER-LU-NAME TO MATCH YOUR CONFIGURATION      *
******************************************************************
     MOVE "NET1.ENLU" TO SI-PARTNER-LU-NAME.
     SET SI-APPLICATION-TP TO TRUE.
     MOVE "CREDRPT " TO SI-TP-NAME.
     MOVE "#INTER" TO SI-MODE-NAME.
     MOVE 124 TO SIDE-INFO-LEN.
     SET SI-SECURITY-NONE TO TRUE.
     CALL "XCMSSI"  USING  TEST-KEY
                    SIDE-INFO-ENTRY
                    SIDE-INFO-LEN
                    CM-RETCODE.
*
     IF CM-OK
       DISPLAY "SIDE-INFO CREATED"
     ELSE
        MOVE "FAILURE TO CREATE SIDE-INFO"
             TO CM-ERROR-DISPLAY-MSG
        PERFORM CLEANUP
           THRU CLEANUP-EXIT
     END-IF.
 SIDE-INITIALIZE-EXIT. EXIT.
*****************************************************************
* CLEAR THE SIDE-INFO CONTROL BLOCK FOR TESTING PURPOSES
* THEN ISSUE DISPLAY REQUEST
*****************************************************************
 SIDE-DISPLAY.
     INITIALIZE SIDE-INFO-ENTRY REPLACING NUMERIC BY 0
                                   ALPHABETIC BY " ".
     DISPLAY "EXTRACTING NEWLY DEFINED SIDE INFORMATION".
     MOVE "CREDRPT" TO SI-SYM-DEST-NAME.
     MOVE 124 TO SIDE-INFO-LEN.
     CALL "XCMESI"  USING TEST-ENTRY-NUMBER
                          SI-SYM-DEST-NAME
                          SIDE-INFO-ENTRY
                          SIDE-INFO-LEN
                          CM-RETCODE.
*
     IF CM-OK THEN
       DISPLAY "-------------------------"
       DISPLAY "SIDE INFORMATION OBTAINED"
       DISPLAY "-------------------------"
       DISPLAY "PARTNER TP NAME = " SI-TP-NAME
       DISPLAY "PARTNER LU NAME = " SI-PARTNER-LU-NAME
       DISPLAY "MODE NAME       = " SI-MODE-NAME
     ELSE
       MOVE "FAILURE DURING SIDE-INFO DISPLAY"
             TO CM-ERROR-DISPLAY-MSG
       PERFORM CLEANUP
       THRU CLEANUP-EXIT
     END-IF.
```

```
  SIDE-DISPLAY-EXIT. EXIT.
************************************************
* DISPLAY EXECUTION COMPLETE OR ERROR MESSAGE *
* NOTE: CREDRPT WILL DEALLOCATE CONVERSATION  *
************************************************
  CLEANUP.
     IF CM-ERROR-DISPLAY-MSG  = SPACES
         DISPLAY "PROGRAM: DEFSIDE EXECUTION COMPLETE"
     ELSE
         DISPLAY "DEFSIDE PROGRAM - ",
                 CM-ERROR-DISPLAY-MSG, " RC= ", CM-RETCODE.
     STOP RUN.
 CLEANUP-EXIT. EXIT.
*****************************************************************
```

DELSIDE.CBL

This program is an example of the function available through the CPI-C extensions provided. *

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.       DELSIDE.
***************************************************************
* THIS PROGRAM IS AN EXAMPLE OF THE FUNCTION AVAILABLE        *
* THROUGH THE CPI-C EXTENSIONS PROVIDED.                      *
*                                                             *
* PURPOSE: DELETE CPI-C SIDE INFORMATION AND DISPLAY RESULT   *
*                                                             *
* INPUT:   SIDE-INFORMATION STRUCTURE.                        *
*                                                             *
* OUTPUT:  CPI-C SIDE INFORMATION TABLE IS DELETED            *
*                                                             *
* NOTE:    FOR THIS SAMPLE PROGRAM, THE KEY FIELD (TEST-KEY), *
*          SUPPORTING THE COMMUNICATIONS MANAGER              *
*          KEYLOCK FEATURE, IS SET TO SPACES.  AS A RESULT,   *
*          THIS PROGRAM WILL RUN SUCCESSFULLY ONLY WHEN       *
*          THE KEYLOCK FEATURE IS NOT SECURED.                *
*                                                             *
***************************************************************
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. PS-2.
 OBJECT-COMPUTER. PS-2.
 SPECIAL-NAMES.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
 I-O-CONTROL.
*
 DATA DIVISION.
 FILE SECTION.
 WORKING-STORAGE SECTION.

 01  TEST-KEY                  PIC  X(1)   VALUE SPACES.

 01  CM-ERROR-DISPLAY-MSG      PIC  X(40)  VALUE SPACES.

**********************************************
* USE THE CPI COMMUNICATIONS PSEUDONYM FILES *
**********************************************
     COPY CMCOBOL.

 LINKAGE SECTION.

 EJECT
*
 PROCEDURE DIVISION.
******************************************************************
**************************  START OF MAINLINE  ******************
******************************************************************
 MAINLINE.

    PERFORM DELETE-SIDE-INFO
       THRU DELETE-SIDE-INFO-EXIT.
     PERFORM CLEANUP
        THRU CLEANUP-EXIT.
     STOP RUN.
```

```
***********************************************
* DELETE SIDE-INFO                            *
***********************************************
  DELETE-SIDE-INFO.
     MOVE "CREDRPT" TO SI-SYM-DEST-NAME.
     CALL "XCMDSI"  USING TEST-KEY
                    SI-SYM-DEST-NAME
                    CM-RETCODE.

    IF CM-OK THEN
      DISPLAY "SIDE INFO DELETED"
    ELSE
      MOVE "FAILURE TO DELETE SIDE-INFO"
            TO CM-ERROR-DISPLAY-MSG
      PERFORM CLEANUP
       THRU CLEANUP-EXIT
    END-IF.
  DELETE-SIDE-INFO-EXIT. EXIT.
***********************************************
* DISPLAY EXECUTION COMPLETE OR ERROR MESSAGE *
***********************************************
  CLEANUP.
    IF CM-OK THEN
      DISPLAY "PROGRAM: DELETE SIDE EXECUTION COMPLETE"
    ELSE
      DISPLAY "DELSIDE PROGRAM - ",
                CM-ERROR-DISPLAY-MSG, " RC= ", CM-RETCODE
      IF  CM-PROGRAM-PARAMETER-CHECK
        DISPLAY "-----------------------------------------------"
        DISPLAY "THIS ERROR CAN RESULT FROM RUNNING DELSIDE"
        DISPLAY "WHEN SIDE INFORMATION HAS ALREADY BEEN DELETED"
        DISPLAY "-----------------------------------------------"
      END-IF
   END-IF.
     STOP RUN.
 CLEANUP-EXIT. EXIT.
*****************************************************************
```

OS/2 REXX Sample Programs

This is a sample of a REXX program to set CPI-C side information.

XCMSSI.CMD

```
/*-------------------------------------------------------*/
/* REXX sample program to set CPI-C side information     */
/*                                                       */
/* Notes: CPICREXX.EXE must be run at some point prior   */
/*        to running this program in order to register   */
/*        the CPICOMM environment to REXX.               */
/*        Also, before running this program, the         */
/*        Communications Manager keylock feature         */
/*        must either be secured with a service key of   */
/*        "svckey" or not be secured at all.             */
/*-------------------------------------------------------*/

/*-----------------------*/
/* set defined constants. */
/*-----------------------*/
CM_OK = 0
XC_APPLICATION_TP = 0
XC_SECURITY_PROGRAM  = 2

say 'CPI-C set side information sample program'

/*-------------------------------------------*/
/* set up parameters and sideinfo structure. */
/*-------------------------------------------*/
sideinfo_len = 124
key = "svckey  "
sideinfo.sym_dest_name = "SYMDEST1"
sideinfo.partner_LU_name = "ALIASLU2"
sideinfo.TP_name =  "MYTPN"
sideinfo.mode_name = "        "
sideinfo.TP_name_type = XC_APPLICATION_TP
sideinfo.conversation_security_type = XC_SECURITY_PROGRAM
sideinfo.security_user_id = "myuserid"
sideinfo.security_password = "xxxxxxxxx"

/*--------------------------------------------------*/
/* issue the CPI-C call to the CPICOMM environment. */
/*--------------------------------------------------*/
address CPICOMM 'xcmssi key sideinfo sideinfo_len retc'

if rc = 0 & retc = CM_OK then
   do
   say '***'
   say 'CPI-C side information successfully set.'
   say '***'
   end
else
   do
   say 'Failure to set CPI-C side information.'
   if rc = 0 then
      say 'CPI-C return code =' retc
   else if rc = 30 then
      say 'CPICREXX has not been executed.'
   else
      say 'REXX return code =' rc
   end
'pause'
'exit'
```

XCMESI.CMD

```
/*-----------------------------------------------------*/
/* REXX sample program to extract CPI-C side information */
/*                                                     */
/* Note: CPICREXX.EXE must be run at some point prior    */
/*       to running this program in order to register   */
/*       the CPICOMM environment to REXX.               */
/*-----------------------------------------------------*/

/*------------------------*/
/* Set defined constants. */
/*------------------------*/
CM_OK = 0
CM_PROGRAM_PARAMETER_CHECK = 24
XC_SECURITY_PROGRAM = 2

say 'CPI-C extract side information sample program'

entry_number = 1
sideinfolen = 124
rc = 0
retc = CM_OK

do while rc = 0 & retc = CM_OK
address cpicomm 'xcmesi entry_number symdest sideinfo sideinfolen retc'
if rc = 0 & retc = CM_OK then
   do
   say '***'
   say '*** Extracted CPI-C side information for entry number:'
        entry_number
   say '***'
   say 'entry_number =' entry_number
   say 'sym_dest_name =' sideinfo.sym_dest_name
   say 'TP_name =' sideinfo.TP_name
   say 'TP_name_type =' sideinfo.TP_name_type
   say 'partner_LU_name =' sideinfo.partner_LU_name
   say 'mode_name =' sideinfo.mode_name
   say 'conversation_security_type =' sideinfo.conversation_security_type
   if sideinfo.conversation_security_type = XC_SECURITY_PROGRAM then
      say 'security_user_id =' sideinfo.security_user_id
   say
   end
else
   do
   if rc = 30 then
      say 'CPICREXX has not been executed.'
   else if rc <> 0 then
      do
      say 'Failure extracting CPI-C side information '
      say 'for entry number:' entry_number
      say 'REXX return code =' rc
      end
   else if retc <> CM_PROGRAM_PARAMETER_CHECK then
      do
      say 'Failure extracting CPI-C side information '
      say 'for entry number:' entry_number
      say 'CPI-C return code =' retc
      end
   else if entry_number = 1 then
      say 'No CPI-C side information.'
   else
      say 'End of CPI-C side information entries.'
```

```
   end
entry_number = entry_number + 1
end

/*---------------------------------------------------------*/
/* Now extract side information by symbolic destination name */
/*---------------------------------------------------------*/
use_symdest = 0
entry_number = use_symdest
symdest = "SYMDEST1"
address cpicomm 'xcmesi entry_number symdest sideinfo sideinfolen retc'

if rc = 0 & retc = CM_OK then
   do
   say '***'
   say '*** Extracted CPI-C side information for symbolic destination name:'
        symdest
   say '***'
   say 'sym_dest_name =' sideinfo.sym_dest_name
   say 'TP_name =' sideinfo.TP_name
   say 'TP_name_type =' sideinfo.TP_name_type
   say 'partner_LU_name =' sideinfo.partner_LU_name
   say 'mode_name =' sideinfo.mode_name
   say 'conversation_security_type =' sideinfo.conversation_security_type
   if sideinfo.conversation_security_type = XC_SECURITY_PROGRAM then
      say 'security_user_id =' sideinfo.security_user_id
   end
else if rc = 0 then
   do
   say 'Failure extracting CPI-C side information'
   say 'for symbolic destination name:' symdest
   say 'CPI-C return code =' retc
   end
else if rc = 30 then
   say 'CPICREXX has not been executed.'
else
   say 'REXX return code =' rc
'pause'
'exit'
```

Chapter 11. CPI Communications on Operating System/400

This chapter summarizes the product-specific information that the OS/400 programmer needs when writing application programs that contain CPI Communications calls.

The information in this chapter should be read in conjunction with the CPI Communications information contained in *AS/400* APPC Programming (SC41-3443)*.

This chapter is organized as follows:

- OS/400 Publications
- OS/400 Operating Environment
  - OS/400 Terms and Concepts
  - Conformance Classes Supported
  - Languages Supported
  - Pseudonym Files
  - Defining Side Information
  - How Dangling Conversations Are Deallocated
  - Scope of the Conversation_ID
  - Identifying Product-Specific Errors
  - Diagnosing Errors
  - When Allocation Requests Are Sent
  - Deviations from the CPI Communications Architecture
- OS/400 Extension Calls
- OS/400 Special Notes

OS/400 Publications

The following OS/400 publications contain detailed product information:

- *AS/400: APPC Programming*, SC41-3443
- *AS/400: Work Management*, SC41-3306
- *AS/400: Communications Management*, SC41-3406
- *AS/400: APPN Support*, SC41-3407
- *AS/400: CL Reference*, SC41-3722
- *AS/400: REXX/400 Programmer's Guide*, SC24-5665

OS/400 Operating Environment

The following sections explain special considerations for use when writing applications for an OS/400 environment.

OS/400 Terms and Concepts

Each piece of work run on the OS/400 system is called a **job**. Each job is a single, identifiable sequence of processing actions that represents a single use of the system. The basic types of jobs performed on the OS/400 system are interactive jobs, batch jobs, spooling jobs, autostart jobs, and prestart jobs.

© Copyright IBM Corp. 1996, 1998

On the OS/400 system, all user jobs operate in an environment called a **subsystem**.

A subsystem is a single, predefined operating environment through which the system coordinates work flow and resource usage. The OS/400 system can contain several independently operating subsystems. The run-time characteristics of a subsystem are defined in an object called a **subsystem description**. IBM supplies several subsystem descriptions that can be used with or without modification:

QINTER Used for interactive jobs

QBATCH Used for batch jobs

QBASE Used for both interactive and communications batch jobs

QCMN Used for communications batch jobs

A new subsystem description can also be defined using the Create Subsystem Description (CRTSBSD) command.

In a subsystem description, **work entries** are defined to identify the sources from which jobs can be started in that subsystem. The types of work entries are as follows:

Autostart job entry
Specifies a job that is automatically started when the subsystem is started.

Workstation entry
Specifies one or a group of workstations from which interactive jobs can be started.

Job queue entry
Specifies one of the job queues from which the subsystem can select **batch jobs**. A batch job is a job that can run independently of a user at a workstation.

Communications entry
Specifies one or a group of communications device descriptions from which communications batch jobs may be started. Communications batch jobs do not use job queues.

Prestart job entry
Identifies an application program to be started to wait for incoming allocation requests.

When a CPI Communications program issues an Allocate (CMALLC) call, the underlying LU 6.2 support sends an allocation request (the LU 6.2 Functional Management Header Type 5, or FMH5). When an allocation request is received by a system, it is called an incoming conversation. Before the OS/400 system can start a job to run the program specified by the incoming conversation, a subsystem must be defined with the appropriate work entries to process this incoming conversation, and the subsystem must be started. The subsystem processing the incoming conversation must have a communications entry defined to identify the communications device and the remote location name (the *partner_LU_name*) on which incoming work can be received. The IBM-supplied subsystem descriptions, QBASE and QCMN, contain default communications entries that can be used for incoming conversations.

Communications entries can be added or modified using the following commands:

- Add Communications Entry (ADDCMNE)
- Remove Communications Entry (RMVCMNE)
- Change Communications Entry (CHGCMNE)

Refer to *OS/400 Work Management (SC41-3306)* and *OS/400 Communications Management (SC41-3406)* for more information on subsystems and communications entries.

Conformance Classes Supported

OS/400 supports the following conformance classes:

- Conversations
- LU 6.2
- Security
- Data Conversion Routines
- Recoverable Transactions

Refer to “Functional Conformance Class Descriptions” on page 746 for a complete description of functional conformance classes.

Languages Supported

The following languages can be used on the OS/400 system to issue CPI Communications calls:

- Application Generator
- ILE* C/400
- ILE COBOL/400
- FORTRAN/400
- REXX/400
- ILE RPG/400

Each language is discussed in the sections that follow.

Application Generator Language: Cross System Product (CSP) is used to implement the Application Generator Common Programming Interface.

The OS/400 system supports CSP/Application Execution but does not support CSP/Application Development. Therefore, CSP application programs cannot be developed on the OS/400 system. However, CSP programs for the OS/400 system can be written in another environment and then run on the OS/400 system.

ILE C/400 Language: The #pragma statement is needed for each CPI Communications routine used in a program. The #pragma statements are included in the ILE C/400 pseudonym file.

The external function names are case-sensitive in ILE C/400 language. On all Version 2 Release 1.1 or earlier OS/400 systems, all CPI Communications call names *must* be typed in uppercase letters. Lowercase letters in CPI Communications call names for programs that issue CPI Communications calls on another platform (for example, cmaccp) must be changed to uppercase letters when using Version 2 Release 1.1 or earlier.

FORTRAN/400 Language: The #pragma statement is needed for each CPI Communications call used in a program. The #pragma statements are included in the FORTRAN/400 pseudonym file.

REXX/400: OS/400 REXX programs are not compiled programs and do not exist as OS/400 objects. To start a REXX program as a result of a program start request, the program name in the program start request must reference a CL program that contains the Start REXX Procedure (STRREXPRC) command.

The Create Command (CRTCMD) command, the Start REXX Procedure (STRREXPRC) command, and the Change Command (CHGCMD) command allow an OS/400 user to specify an initial command environment to be used when an ADDRESS statement is not coded and a command is encountered. The special value *CPICOMM can be used for the command environment keyword CMDENV on the STRREXPRC command to specify that CPI Communications is the initial command environment. The special value *CPICOMM can be used for the REXX command environment keyword REXCMDENV on the CRTCMD and CHGCMD commands to specify that CPI Communications is the initial command environment.

Pseudonym Files

Application Generator Language: CSP programs for the OS/400 system can be written in another environment and run on the OS/400 system. For this reason, no pseudonym file is provided for CSP on the OS/400 system.

ILE C/400 Language: The #pragma statements are included in the pseudonym file provided by IBM. The pseudonym file resides in library QSYSINC , file H, member CMC.

COBOL/400 Language: Pseudonym files are provided for CPI Communications programs written in the COBOL/400 and ILE COBOL/400 languages. The pseudonym file for COBOL/400 resides in library QSYSINC, file QLBLSRC, member CMCOBOL. The pseudonym file for ILE COBOL/400 resides in library QSYSINC, file QCBLLESRC, member CMCOBOL.

FORTRAN/400 Language: The #pragma statements are included in the pseudonym file provided by IBM. The pseudonym file resides in library QFTN, file QIFOINC, member CMFORTRN.

RPG/400 Language: Pseudonym files are provided for CPI Communications programs written in the RPG/400 and ILE RPG/400 languages. The pseudonym file for RPG/400 resides in library QSYSINC, file QRPGSRC, member CMRPG. The pseudonym file for ILE RPG/400 resides in library QSYSINC, file QRPGLESRC, member CMRPG.

REXX/400: No pseudonym file is provided for the OS/400 REXX language.

Defining Side Information

For a program to establish a conversation with a partner program, CPI Communications requires initialization parameters, such as the name of the partner program and the name of the LU at the node of the partner program. These initialization parameters are stored in the side information. On the OS/400 system, the side information is referred to as **communications side information**.

A program must specify the communications side information name as the symbolic destination name (*sym_dest_name*) parameter on the Initialize_Conversation call to use the stored characteristics. If a program does not specify a side information name (that is, the *sym_dest_name* is eight space characters), it must issue the Set_Partner_LU_Name and Set_TP_Name calls. Also, when no side information name is specified, the *mode_name* characteristic for the conversation defaults to eight space characters. To override this default, the program must issue the Set_Mode_Name call.

On the OS/400 system, the communications side information is an object of type *CSI (communications side information). It contains information that defines the remote system (for example, the remote location name, remote network identifier, and mode). The OS/400 communications side information object also contains additional information, namely, the device description, the local location name (local network LU name), and the authority. The remote network ID, remote location name, device description, and local location name are used by the OS/400 system to connect to the remote system.

Note: On the OS/400 system, the conversation security type, user ID, and password cannot be stored in the communications side information. To set these conversation characteristics, use the following calls:

- Set_Conversation_Security_Password (CMSCSP)
- Set_Conversation_Security_Type (CMSCST)
- Set_Conversation_Security_User_ID (CMSCSU)

Managing the Communications Side Information

To manage the side information object, the OS/400 system provides Control Language (CL) commands that can be used to create, display, print, change, delete, and work with the communications side information. Command prompting from a display is also provided to perform these tasks. The following is a list of the CL commands that can be used to manage the communications side information object:

OS/400	Command Description
CRTCSI	Used to create the CPI Communications side information object
CHGCSI	Used to change the CPI Communications side information object
DSPCSI	Used to display or print the CPI Communications side information object
DLTCSI	Used to delete the CPI Communications side information object
WRKCSI	Provides a menu from which the user can create, change, display, delete, or print the CPI Communications side information object

The following table describes the information contained in the communications side information object and maps it to the OS/400 CPI Communications parameters.

Table 41. Description of OS/400 Communications Side Information Object

OS/400 System Parameters	Description
RMTLOCNAME and RMTNETID	The remote location name (RMTLOCNAME) and remote network ID (RMTNETID) make up the name of the logical unit (LU) on the remote system. These parameters correspond to the *partner_LU_name*, which is defined by the CPI Communications architecture as a required characteristic for the side information. A fully qualified *partner_LU_name* is defined as the network ID concatenated by a period with the network LU name (that is, `network ID.network LU name`). On the OS/400 system, the remote network ID is the network ID, and the remote location name is the network LU name.
MODE	The name of the mode used to control the session. It is used to designate the properties for the session that will be allocated for the conversation, such as the class of service to be used on the conversation. This parameter corresponds to the CPI Communications *mode_name* characteristic. To use a *mode_name* of eight space characters, the special value of BLANK must be used. The OS/400 system does not support sending a mode name of 'BLANKxxx' to a remote system, where x is a space character.
DEV	The name of the device description, which describes the characteristics of the logical connection between a local and remote location. This parameter is specific to the OS/400 system and further qualifies the connection defined by the remote location name and the remote network ID.
LCLLOCNAME	The name of the local location, which specifies the local network LU name in a network. The OS/400 environment supports multiple local location names. This parameter is specific to the OS/400 system and further qualifies the connection defined by the remote location name and the remote network ID.
PGMNAME	The name of the target program that is to be started. This corresponds to the *TP_name*, which is defined by the CPI Communications architecture as a required characteristic for the communications side information.
AUT	The authority given to users who do not have specific authority to the communications side information object. This parameter is specific to the OS/400 system.

How Dangling Conversations Are Deallocated

When a CPI Communications program ends before one of its conversations is deallocated, the conversation is considered to be a dangling conversation. The OS/400 CPI Communications support ends dangling conversations when a *job* in which the program was running ends. When the OS/400 CPI Communications support ends a dangling conversation for a job, a *return_code* of CM_DEALLOCATED_ABEND is sent to the partner program, and a message indicating that the conversation has ended is placed in the system history log. The Display Log (DSPLOG) command can be used to display or print the system history log.

Reclaim Resource Processing

A conversation is considered a resource on the OS/400 system. Therefore, the Reclaim Resource (RCLRSC) command can be used by a user or by a program to end conversations on the OS/400 system. For each conversation ended by the reclaim resource processing, a *return_code* of CM_DEALLOCATED_ABEND is sent to the partner program, and a message indicating that the conversation has ended is placed in the system history log. The Display Log (DSPLOG) command can be used to display or print the system history log.

OS/400 CL Reference (SC41-3722) contains more information on the RCLRSC command.

Scope of the Conversation_ID

The OS/400 CPI Communications support associates conversations with OS/400 jobs. Each conversation is assigned a *conversation_ID* that is unique within the job; however, the *conversation_ID* is not a system-wide unique value. Therefore, a *conversation_ID* in one job cannot be accessed by another job.

Identifying Product-Specific Errors

This section discusses errors that can be returned on calls to CPI Communications routines because of reasons and errors that are specific to the OS/400 system.

CM_PRODUCT_SPECIFIC_ERROR

The CM_PRODUCT_SPECIFIC_ERROR *return_code* is returned on the following calls for the designated reasons:

Set_Mode_Name (CMSMN)

OS/400 CPI Communications only supports the use of the following OS/400 special values when they are specified in the side information. These special values cannot be specified on a Set_Mode_Name call.

- OS/400 special value *NETATR was specified for *mode_name*.
- OS/400 special value BLANK was specified for *mode_name*.
- An unexpected OS/400 internal error occurred.

Note: The *conversation_state* does not change for this *return_code* on this call.

Set_Partner_LU_Name (CMSPLN)

OS/400 CPI Communications only supports the use of the following OS/400 special values when they are specified in the side information. These special values cannot be specified on a Set_Partner_LU_Name call.

- OS/400 special value *LOC was specified for the network ID portion of the *partner_LU_name*.
- OS/400 special value *NETATR was specified for the network ID portion of the *partner_LU_name*.
- OS/400 special value *NONE was specified for the network ID portion of the *partner_LU_name*.
- An unexpected OS/400 internal error occurred.

Note: The *conversation_state* does not change for this *return_code* on this call.

Accept_Conversation (CMACCP)
The conversation enters **Reset** state when CM_PRODUCT_SPECIFIC_ERROR is returned in a *return_code* for the following reasons:

- The program is defined in a prestart job entry (ADDPJE command) and is being ended.
- The program has already issued an ACQUIRE operation to the *REQUESTER device using an Intersystem Communications Function (ICF) file or a communications file or a mixed file.
- OS/400 CPI Communications was unable to obtain enough system storage to support the conversation.
- An unexpected OS/400 internal error occurred.

Allocate (CMALLC)
The conversation enters **Reset** state when CM_PRODUCT_SPECIFIC_ERROR is returned in the *return_code* for this call. CM_PRODUCT_SPECIFIC_ERROR can be returned for the following reasons:

- The program is not authorized to use the APPC device description that was selected based on the *partner_LU_name* and the side information location parameters.
- The program specified a *partner_LU_name* that caused the system to select a device description that is not an APPC device. (The network LU name portion of the *partner_LU_name* specified an OS/400 remote location name that is configured in a non-APPC device description.)
- The *partner_LU_name* requested a conversation with a network LU name that resides on the same control point, and APPN support is being used. (The OS/400 system does not support communications between two network LU names on the same control point when APPN support is used.)
- An unexpected OS/400 internal error occurred.

Initialize_Conversation (CMINIT)
The conversation enters **Reset** state when CM_PRODUCT_SPECIFIC_ERROR is returned in a *return_code* for the following reasons:

- OS/400 CPI Communications was unable to obtain enough system storage to support the conversation.
- An unexpected OS/400 internal error occurred.

Any other set or any extract call
An unexpected OS/400 internal error occurred. The state of the conversation remains unchanged.

Diagnosing Errors

Each time OS/400 CPI Communications support returns an error *return_code* to a program, a message is also placed in the job log for the user's job to indicate the cause of the error. The Display Job Log (DSPJOBLOG) command can be used to display or print the job log for any OS/400 job.

OS/400 CPI Communications Support of Log_Data

A CPI Communications program may set *log_data* using the Set_Log_Data (CMSLD) call on a basic conversation. If a program has set *log_data*, the *log_data* will be sent to the partner system when one of the following occurs:

- The CPI Communications program issues a Send_Error (CMSERR) call.
- The CPI Communications program issues a Deallocate (CMDEAL) call with *deallocate_type* of CM_DEALLOCATE_ABEND.

When the OS/400 sends *log_data* to a partner system, it places a message in the system history log. This message includes the *log_data* that was sent.

When an OS/400 receives *log_data* from a partner system, it places a message in the system history log. This message includes the *log_data* that was received.

The OS/400 system history log can be viewed or printed using the Display Log (DSPLOG) command.

Return Codes

The following *return_code* values can be returned on calls to CPI Communications routines because of reasons and errors that are specific to the OS/400 operating system. Many of these same errors can occur when running CPI Communications over TCP/IP (AnyNet support) because APPC over TCP/IP controllers simulate APPN support.

CM_ALLOCATE_FAILURE_RETRY

- The operator varied off the APPC device, or the APPC device was in a recovery mode and the recovery was cancelled via a command or via an answer of cancel to a recovery message on the system operator message queue.
- The OS/400 APPN support attempted to dynamically vary on the APPC device description needed for the *partner_LU_name*. A line failure or station failure may have occurred during APPN processing, and the recovery was cancelled.
- The APPC device description needed for the *partner_LU_name* is a dependent device configured to use APPN support, but the device is varied off. OS/400 APPN support does not dynamically vary on dependent devices.
- The OS/400 APPN support could not determine an available route to the destination specified by the *partner_LU_name*. For example, directory services encountered a link failure on a control point in a control point session.
- The exchange log name processing failed.
- The protected conversation registration failed.

CM_ALLOCATE_FAILURE_NO_RETRY

- The OS/400 APPN support attempted to dynamically create the APPC device description needed for the *partner_LU_name*. The attempt was unsuccessful because of a previous user configuration error.
- The OS/400 APPN support attempted to dynamically add the mode needed for the *mode_name* to the APPC device description needed for

the *partner_LU_name*. The attempt was unsuccessful because the needed device description already has the maximum number of modes that is allowed by configuration services.

- The OS/400 APPN support attempted to start the mode needed for the *mode_name*. The attempt was not successful because of a change-number-of-sessions (CNOS) failure.
- The OS/400 APPN support encountered a route calculation error.
- The class of service (COS) specified in the mode description cannot be found. The mode description is specified by the *mode_name* conversation characteristic.
- No route exists that satisfies the COS characteristic specified by the COS parameter in the mode description. The mode description is specified by the *mode_name* conversation characteristic.
- The local network LU (local location name) that was specified in the side information cannot be found.
- The device or devices specified single-session support.
- A protected conversation could not be established because the program was running in the System/36 or System/38 environment.

CM_PARAMETER_ERROR
The CM_PARAMETER_ERROR can be returned on the Allocate call (CMALLC) for the following reasons:

- The requested *partner_LU_name* does not reside on this end node, and no network node is available to search for the *partner_LU_name*.
- The requested *partner_LU_name* cannot be located by APPN directory services.

REXX Reserved RC Variable

The OS/400 CPICOMM environment support returns the following values in the REXX RC variable:

Code	Meaning
0	The CPI Communications routine was successfully called.
-3	The routine name specified does not exist or was spelled incorrectly.
-9	Insufficient storage is available. Attempt the call again when more storage is available.
-10	Too many parameters were specified for the CPI Communications calls. Refer to the detailed description for the specified call in this book to find the proper number of parameters.
-11	Not enough parameters were specified for the CPI Communications call. Refer to the detailed description for the specified call in this book to find the proper number of parameters.
-14	An internal system error occurred in the CPICOMM environment. Attempt the call again. If the condition continues, report the problem using the Analyze Problem (ANZPRB) command.

-24 An unexpected error occurred on an internal fetch variable contents call. Attempt the call again. If the condition continues, report the problem using the Analyze Problem (ANZPRB) command.

-25 An unexpected error occurred on an internal set variable contents call. Attempt the call again. If the condition continues, report the problem using the Analyze Problem (ANZPRB) command.

-27 The value of the module call parameter is not valid. This value cannot be converted to the binary integer format required by CPI Communications. The value was more than 31 digits, contained a digit that was not 0–9, a plus sign (+), a minus sign (-), or contained an embedded blank.

-28 An invalid variable name was used. Refer to *AS/400: REXX/400 Programmer's Guide* (SC24-5665) for information concerning valid variable names.

-30 A character string that started with a shift-out character did not end with a shift-in character. The command passed to the REXX CPICOMM environment must contain valid bracketed DBCS characters.

REXX Error and Failure Conditions

Conditions are problems or other occurrences that may arise while a REXX program is running. **Condition traps** are routines that take control when the specified conditions are met. A condition trap is enabled by using the SIGNAL ON or CALL ON instructions.

The programmer must be aware of two conditions in the OS/400 CPI Communications command environment: FAILURE and ERROR. The OS/400 CPI Communications command environment indicates a FAILURE condition when a negative value is returned in the RC variable. If the program has enabled a condition trap for the FAILURE condition, control passes to the routine that is named to handle the FAILURE condition. If the program has not enabled a condition trap for a FAILURE condition but has enabled a condition trap for an ERROR condition, control passes to the routine that is named to handle the ERROR condition.

Note: An ERROR condition trap will never receive control from the CPI Communications command environment if a FAILURE condition trap is also enabled.

Refer to *AS/400: REXX/400 Programmer's Guide* (SC24-5665) for information concerning the error and failure conditions and how to process them.

Tracing CPI Communications

Use the Trace Common Programming Interface Communications (TRCCPIC) command to capture information about CPI Communications calls that are being processed by the application program. This trace information can be collected in a current job or in a job being serviced by the Start Service Job (STRSRVJOB) command. Tracing CPI Communications can be done before running a job or after a job is active. For more information about tracing CPI Communications, see *OS/400 Communications Management (SC41-3406)*.

When Allocation Requests Are Sent

OS/400 CPI Communications support sends the allocation request before returning control to the program after the Allocate call.

OS/400 Extension Calls

OS/400 provides no CPI Communications extension calls.

OS/400 Special Notes

This section discusses TCP/IP support, prestarting jobs, multiple conversation support, and portability considerations that the programmer should be aware of when writing OS/400 CPI Communications application programs.

CPI Communications over TCP/IP Support

OS/400 CPI Communications applications programs can use TCP/IP support for communications with no changes. See *OS/400 APPC Programming* (SC41-3443) and *OS/400 Communications Configuration* (SC41-3401) for more information.

Prestarting Jobs for Incoming Conversations

To minimize the time required for a program to accept a conversation with its partner program, an OS/400 prestart job entry can be used. When a prestart job entry is used, the application program is started before an allocation request is received from the partner program. Each prestart job entry contains a program name, library name, user profile, and other attributes that the subsystem uses to create and manage a pool of prestart jobs.

The following must be done to use a prestart job entry:

1. Define a prestart job entry. A prestart job entry is defined, using the Add Prestart Job Entry (ADDPJE) command, in the subsystem that contains the communications entry.
2. Start the prestart job entry. The prestart job entry can be started at the same time the subsystem is started or the Start Prestart Jobs (STRPJ) command can be used.

Programs should be designed with the following considerations when a prestart job entry is used:

- To ensure that the initial processing is completed before the allocation request is received, a prestart job program should do as much work as possible (for example, opening database files) before issuing the Accept_Conversation (CMACCP) call.

  The Accept_Conversation call will not complete until an allocation request is received for the application program. When this request is received, the application program receives control with a *return_code* of CM_OK (assuming that no authorization or other problems are encountered), and the application program can immediately begin processing.

- When a prestart job program has finished servicing an incoming conversation, the conversation enters the **Reset** state, and the program may then make itself

available for another incoming conversation by issuing another Accept_Conversation call.

- Only resources that are used specifically for a conversation should be deallocated. For example, if a database file is used for most conversations, there is no need to close the file and then open it each time a conversation is deallocated and a new conversation is accepted.
- Prestart jobs can be used for protected conversations, but there are some considerations:
  - If a prestart job attempts an Accept_Conversation (CMACCP) call and the prestart job has already allocated protected conversations, the CMACCP call is rejected. The CMACCP call returns a CM_PRODUCT_SPECIFIC_ERROR return code, and a message is sent to the job log.
  - When protected conversations are already active, a prestart job must end them before it can accept an incoming, protected conversation.
  - A prestart job can accept an incoming, protected conversation if there are unprotected conversations already active.

Programs that are designed to use prestart job entries may not be portable to other system environments.

Refer to *OS/400 APPC Programming (SC41-3443)* for more information about using CPI Communications with prestart job entries.

Multiple Conversation Support

Application programs running in an OS/400 job can allocate many conversations and communicate concurrently over these conversations. However, only one incoming conversation can exist for one OS/400 job at any time. In addition, only prestart jobs can accept subsequent incoming conversations after deallocating an initial incoming conversation.

For example, job A can accept, process, and deallocate a conversation. If job A is a prestart job, it can then issue another Accept_Conversation to wait for another incoming conversation to process. If job A is not a prestart job, it cannot process another incoming conversation.

Portability Considerations

The following are portability considerations for CPI Communications application programs:

- To maintain system security, the conversation security type, user ID, and password cannot be stored in the communications side information. To set these conversation characteristics, use the following calls:
  - Set_Conversation_Security_Password (CMSCSP)
  - Set_Conversation_Security_Type (CMSCST)

    OS/400 supports these values for Set_Conversation_Security_Type:
    - CM_SECURITY_NONE
    - CM_SECURITY_SAME
    - CM_SECURITY_PROGRAM
    - CM_SECURITY_PROGRAM_STRONG
  - Set_Conversation_Security_User_ID (CMSCSU)

- Programs that issue Extract_Partner_LU_Name (CMEPLN) or Extract_Mode_Name (CMEMN) calls before allocating the conversation with the Allocate (CMALLC) call may need to be modified. This is because the OS/400 CPI Communications support can also return the OS/400 special values specified in the communications side information (*NETATR and *LOC, for example). Once the Allocate call is successfully issued, these special values are resolved by the CPI Communications support and are no longer returned.

  For example, if a network LU name of NEWYORK exists in a network called APPN, the OS/400 communications side information could specify a network ID of *NETATR. This means that the name of the local network will be used. Therefore, a CMEPLN call that is issued *before* the Allocate call could return *NETATR.NEWYORK. However, a CMEPLN call issued *after* the Allocate call could now return APPN.NEWYORK if APPN is the local network identifier in the network attributes. Refer to *OS/400 APPC Programming (SC41-3443)* for more information on these special values.

- Programs that handle multiple incoming conversations (one at a time) might not be portable. These programs are designed to issue an Accept_Conversation call multiple times to use the prestart job entry function.

Chapter 12. CPI Communications on VM/ESA CMS

This appendix summarizes the product-specific information that the VM programmer needs when writing application programs that contain CPI Communications calls or VM/ESA extension calls.

This appendix contains information about VM/ESA's implementation of and extensions to CPI Communications. It describes the CPI Communications extension calls that can be used in VM/ESA to take advantage of VM/ESA's capabilities; note, however, that a program using any of these VM/ESA extension calls may require modification if moved to another SAA operating system.

Before reading this appendix, the reader should be familiar with the connectivity programming section of the *VM/ESA: CMS Application Development Guide*. That book discusses connectivity terminology used in this appendix, gives an overview of communications programming on VM/ESA, introduces the use of CPI Communications on VM/ESA, and describes work units and logical units of work. It also contains scenarios and example programs that help demonstrate how to use CPI Communications in VM/ESA. In addition, readers unfamiliar with communications programming may also find the *VM/ESA: Common Programming Interface Communications User's Guide* helpful.

This appendix is organized as follows:

- VM/ESA Publications
- VM/ESA Operating Environment
  - Conformance Classes Supported
  - Languages Supported
  - Pseudonym Files
  - Defining Side Information
  - How Dangling Conversations Are Deallocated
  - Scope of the Conversation_ID
  - Identifying Product-Specific Errors
  - Diagnosing Errors
  - When Allocation Requests Are Sent
  - Deviations from the CPI Communications Architecture
- VM/ESA Extension Calls
- VM/ESA Special Notes

VM Publications

The following VM/ESA publications contain detailed product information:

- *VM/ESA: CMS Application Multitasking*, SC24-5652
- *VM/ESA: CMS Command Reference*, SC24-5461
- *VM/ESA: CMS Application Development Reference*, SC24-5451
- *VM/ESA: CMS Application Development Reference for Assembler*, SC24-5453
- *VM/ESA: CMS Application Development Guide*, SC24-5450
- *VM/ESA: CMS User’s Guide*, SC24-5460
- *VM/ESA: CMS Application Development Guide for Assembler*, SC24-5452
- *VM/ESA: Planning and Administration*, SC24-5521

© Copyright IBM Corp. 1996, 1998

- *VM/ESA: Connectivity Planning, Administration, and Operation*, SC24-5448
- *VM/ESA: CP Programming Services*, SC24-5520
- *VM/ESA: System Messages and Codes*, SC24-5529
- *VM/ESA: REXX/VM Reference*, SC24-5466
- *VM/ESA: Common Programming Interface Communications User's Guide*, SC24-5595
- *VM/ESA: SFS and CRR Planning, Administration, and Operation*, SC24-5649
- *VM/ESA: Conversion Guide and Notebook for VM/XA SP and VM/ESA*, SC24-5525

VM/ESA Operating Environment

The following sections explain some special considerations that should be understood when writing applications for a VM/ESA environment.

Conformance Classes Supported

VM/ESA supports the following conformance classes:

- Conversations

  All conversations, with the exception of Extract_Maximum_Buffer_Size (CMEMBS)

- LU 6.2

Refer to "Functional Conformance Class Descriptions" on page 746 for a complete description of functional conformance classes.

Languages Supported

The following SAA languages can be used on VM/ESA to issue CPI Communications calls and VM/ESA extension calls:

- Application Generator (Cross System Product implementation)
- C
- COBOL
- FORTRAN
- PL/I
- REXX (SAA Procedures Language).

In addition, the following non-SAA languages can be used on VM/ESA:

- Assembler
- Pascal.

The following list shows the call syntax for Assembler and Pascal:

Assembler

```
CALL routine_name,(parm1,parm2,...return_code),VL
```

Pascal

```
routine_name (parm1,parm2,...return_code);
```

Programming Language Considerations

This section describes the programming considerations a programmer should keep in mind when writing and running programs that use CPI Communications in a VM/ESA environment. Specific notes for Application Generator, Assembler, C, Pascal, and REXX are listed in the sections that follow.

Note: For all languages except REXX, when the CPI Communications application is bound into a module through the usual CMS LOAD, INCLUDE, GENMOD, or LKED procedures, the disk or directory containing CMSSAA TXTLIB and VMLIB TXTLIB should be accessed and CMSSAA and VMLIB should be specified on a GLOBAL TXTLIB command. CMSSAA TXTLIB is not referenced at run time.

Application Generator: Cross System Product (CSP) is the implementing product for the Application Generator Common Programming Interface.

Please keep the following note in mind when coding a CSP program that issues CPI Communications calls:

- Use the ‘NONCSP’ parameter in the CALL statement to avoid searching the application's load file for the CPI Communications routines.

Assembler: Please keep the following note in mind when coding an assembler program that issues CPI Communications calls:

- When building the parameter list to pass to CPI Communications, make sure the high-order bit of all the addresses in the parameter list is zero, except for the last entry in the parameter list. The last entry must have the high-order bit set to designate the end of the parameter list. Specifying `VL` on the routine call as shown on page 528 causes the high-order bit of the last address parameter to be set to 1.

C: Please keep the following notes in mind when coding a C program that issues CPI Communications calls:

- VM/ESA does not put a terminating null byte in character strings it returns. C programs must take this into consideration.
- The `#pragma linkage` statement is needed for each CPI Communications call used in a program. The `#pragma` statements are included in the CMC COPY pseudonym file provided with VM/ESA. Note that because C is a case-sensitive language, the CPI Communications call name must be coded exactly as specified in the `#pragma` statement.
- If the C/370 licensed program is being used, the following REXX exec may set up the proper environment for compiling C programs:

```
/* A REXX exec to set the loader tables and perform necessary
   global commands */
'SET LDRTBLS 8'
'GLOBAL LOADLIB EDCLINK'
'GLOBAL TXTLIB EDCBASE IBMLIB CMSLIB CMSSAA VMLIB'
exit
```

Pascal: Please keep the following notes in mind when coding a Pascal program that issues CPI Communications calls:

- The %INCLUDE for CMPASCAL should be a constant declaration.

- Use internal procedure statements for each CPI Communications call being used, and declare each call as a FORTRAN call.
- Parameters should be passed as variables by reference, rather than passing them as literals and constants.
- String parameters must have a length specified.
- Use PASCMOD, not LOAD, to build a load module. Enter the following commands to compile, load, and run an executable Pascal program named 'myprog':

```
VSPASCAL myprog (lib(dmsgpi))              /* compile 'myprog' */
PASCMOD myprog CMSSAA VMLIB      /* build module file, make sure
                          CMSSAA TXTLIB and VMLIB TXTLIB are used */
myprog                                          /* run 'myprog' */
```

REXX (SAA Procedures Language): Please keep the following notes in mind when coding a REXX program that issues CPI Communications calls:

- If Send_Data (CMSEND) is called from a REXX program, the *buffer* parameter specified on the call cannot contain more than 32767 bytes of data. A data field exceeding this size must be partitioned into units of 32767 bytes or less. This restriction applies only to REXX.
- The special REXX variable RC should be checked following each CPI Communications call before processing any values returned by the call. If the RC value is not zero, the output parameters of the call are not meaningful.

Pseudonym Files

The pseudonym files provided with VM/ESA include all of the CPI Communications pseudonym values listed in Table 59 on page 642 along with the additional VM/ESA values shown in Table 45 on page 576.

By including the appropriate file, a program can use pseudonyms for the actual CPI Communications integer values. Table 42 lists the various pseudonym files and shows where they reside in VM/ESA.

Table 42. Summary of CPI Communications Pseudonym Files

Language	File Name	File Location
Application Generator	CMCSP COPY	System disk
Assembler H	CMHASM COPY	DMSGPI MACLIB
C	CMC COPY	DMSGPI MACLIB
COBOL	CMCOBOL COPY	DMSGPI MACLIB
FORTRAN	CMFORTRN COPY	DMSGPI MACLIB
Pascal	CMPASCAL COPY	DMSGPI MACLIB
PL/I	CMPLI COPY	DMSGPI MACLIB
REXX	CMREXX COPY	System disk

The following notes discuss some special considerations when accessing the pseudonym files:

Assembler: The following example shows how to use the CMHASM COPY file. Before assembling it, enter the command `GLOBAL MACLIB DMSGPI OSMACRO` to get access to CMHASM COPY (in DMSGPI MACLIB) and the CALL macro (in OSMACRO MACLIB).

```
MYPROG   CSECT
*-------------------------------------------------------------------*
* Standard OS linkage
*-------------------------------------------------------------------*
         STM   R14,R12,12(R13)    Save system's registers
         USING MYPROG,R12         Establish base register 12
         ST    R13,SAVEMAIN+4     Save pointer to system's save area
         LA    R13,SAVEMAIN       R13 points to our save area
*-------------------------------------------------------------------*
* Get addressability to the CMHASM file
*-------------------------------------------------------------------*
         USING CMHASM,R8
         L     R8,=V(CMHASM)
*-------------------------------------------------------------------*
* Do Initialize conversation
*-------------------------------------------------------------------*
         CALL  CMINIT,(CONID,SYMDNAME,RETCODE),VL
         L     R4,RETCODE          Get the return code
*-------------------------------------------------------------------*
* Compare return code from CMINIT with CM_OK value in CMHASM COPY
*-------------------------------------------------------------------*
         C     R4,CM_OK            Did CMINIT work successfully?
         BZ    INITOK              Yes, branch and handle it.
*-------------------------------------------------------------------*
* Handle CMINIT error
*-------------------------------------------------------------------*
         :
INITOK   EQU   *
*-------------------------------------------------------------------*
* CMINIT worked fine
*-------------------------------------------------------------------*
         :
         COPY  CMHASM             Include for CMHASM
         END   MYPROG
```

C: Users who do not have access to Version 2 of the IBM C/370 Compiler and Library, which provides MACLIB support in VM/ESA, may have to copy the CMC COPY file from DMSGPI MACLIB to a minidisk or directory. One way to do this is by entering

```
XEDIT DMSGPI MACLIB (MEMBER CMC
```

and then filing it as CMC COPY on a read/write minidisk or directory.

When using Version 2 of the IBM C/370 Compiler and Library, the following example shows how to use the MACLIB support on VM/ESA when compiling a program called ‘myprog’:

```
CC myprog (SEARCH((CMC.COPY)=(LIB(DMSGPI)))
```

REXX (SAA Procedures Language): The following example shows how to use the CMREXX COPY file to equate pseudonyms to their integer values:

```
/*-------------------------------------------------------------------*/
/* Equate pseudonyms to integer values based on CMREXX COPY file.     */
/*-------------------------------------------------------------------*/
address command 'EXECIO * DISKR CMREXX COPY * (FINIS STEM PSEUDONYM.'
do index = 1 to pseudonym.0
  interpret pseudonym.index
end
```

CMS Pipelines provides an alternative to the EXECIO statement that was used in the previous example:

```
/*-------------------------------------------------------------------*/
/* Equate pseudonyms to integer values based on CMREXX COPY file.     */
/*-------------------------------------------------------------------*/
address command 'PIPE < CMREXX COPY * | STEM PSEUDONYM.'
do index = 1 to pseudonym.0
  interpret pseudonym.index
end
```

Defining Side Information

CPI Communications defines side information, which is a set of values used when starting conversations. VM/ESA extends the side information to include access security information.

VM/ESA implements side information using CMS communications directory files. A communications directory file is a CMS NAMES format file that can be set up on a system level (by a system administrator) or on a user level. The default name for the system-level communications directory file is SCOMDIR NAMES and the default name for the user-level file is UCOMDIR NAMES.

Note: Communications directories can be created or changed using the NAMES command. See the NAMES command usage notes in the *VM/ESA: CMS Command Reference* for more information.

Table 43 on page 533 lists and describes the tags that can be used in a CMS communications directory.

<table>
<tr><td colspan="4">Table 43. Contents of a CMS Communications Directory File</td></tr>
<tr><th>Tag</th><th colspan="3">What the Value on the Tag Specifies</th></tr>
<tr><td>:nick.</td><td colspan="3">The symbolic destination name of up to 8 characters for the target resource.</td></tr>
<tr><td rowspan="7">:luname.</td><td colspan="3">The partner LU name (locally known LU name) that identifies where the resource resides. This name consists of two fields of up to 8 characters each separated by at least one blank. The fields are an LU name qualifier (network ID) and a target LU name. The values that can be used for each depend on the connection:</td></tr>
<tr><th>Connection</th><th>LU Name Qualifier</th><th>Target LU name</th></tr>
<tr><td>To private resource within the TSAF or CS collection</td><td></td><td>private resource manager's user ID</td></tr>
<tr><td>To a local or system resource, or to a global resource within the TSAF or CS collection (tag may be omitted)</td><td></td><td>blank</td></tr>
<tr><td>Outside the TSAF or CS collection</td><td>VM gateway name</td><td>name of target LU</td></tr>
<tr><td>To global or system resource on a particular system in the CS or TSAF collection</td><td>system gateway name of target system</td><td>blank</td></tr>
<tr><td>To a private resource on a particular system in the CS or TSAF collection</td><td>system gateway name of the target system</td><td>private resource manager's user ID</td></tr>
<tr><td>:tpn.</td><td colspan="3">The transaction program name as it is known at the target LU for connections in the local VM system or collection. For a local or global resource, this is the resource name identified by the resource manager. For a private resource, this is the nickname specified in the private resource server virtual machine's $SERVER$ NAMES file. For a target LU in the SNA network, this is the transaction program name. The transaction program name cannot start with a period. '&TSAF' is the transaction program name reserved for TSAF virtual machines that are using APPC links.</td></tr>
<tr><td>:modename.</td><td colspan="3">For connections outside the TSAF or CS collection, this field specifies the mode name for the SNA session connecting the gateway to the target LU. For connections within the TSAF or CS collection, this field specifies a mode name of either VMINT or VMBAT, or it is omitted. Only user programs running in requester virtual machines with OPTION COMSRV specified in their CP directory entry can specify connections with a mode name of VMINT or VMBAT.</td></tr>
<tr><td>:security.</td><td colspan="3">The access security type of the conversation (NONE, SAME, or PGM).</td></tr>
<tr><td>:userid.[7]</td><td colspan="3">The access security user ID. (This is used for security type PGM and is ignored for other security types.)</td></tr>
<tr><td>:password.[7]</td><td colspan="3">The access security password. (This is used for security type PGM and is ignored for other security types.)</td></tr>
</table>

See *VM/ESA: Connectivity Planning, Administration, and Operation* for examples of communications directory entries and information on using them to implement or modify side information. Once a communications directory file has been created or modified, it must be put into effect by entering the SET COMDIR command. (Refer to the *VM/ESA: CMS Command Reference* for details on this command.) The VM/ESA-supplied SYSPROF EXEC automatically attempts to load SCOMDIR NAMES and UCOMDIR NAMES during CMS initialization.

7 Including access security user IDs and passwords in a CMS communications directory is a potential security exposure. Security user IDs and passwords can be specified on the APPCPASS statement in the source virtual machine's directory, rather than in this file. *VM/ESA: Connectivity Planning, Administration, and Operation* explains this in detail.

How Dangling Conversations Are Deallocated

A program should terminate all conversations before the end of the program. However, if the program does not terminate all conversations, node services will deallocate them abnormally during CMS end-of-work unit processing. These leftover conversations are referred to as *dangling* conversations.

On VM/ESA, node services will deallocate all dangling conversations with APPCVM SEVER specifying a sever code of DEALLOCATE_ABEND_SVC. This applies to both mapped and basic conversations. The DEALLOCATE_ABEND_SVC sever code indicates to the partner program that node services issued the deallocate.

The return code that is reflected to the CPI Communications conversation partner depends on the *conversation_type* and *sync_level* characteristics. When the *sync_level* is CM_SYNC_POINT, the return code indicates that a backout (BO) is required. If the *conversation_type* is CM_BASIC_CONVERSATION, the partner program sees a return code of either:

- CM_DEALLOCATED_ABEND_SVC
- CM_DEALLOCATED_ABEND_SVC_BO.

If the *conversation_type* is CM_MAPPED_CONVERSATION, the partner program sees a return code of either:

- CM_RESOURCE_FAILURE_NO_RETRY
- CM_RESOURCE_FAIL_NO_RETRY_BO.

In addition, at CMS end of command, conversations left in **Initialize** state are deallocated.

If a conversation that attempts to invoke a private resource manager is never accepted, the connection request is severed and a return code of CM_TPN_NOT_RECOGNIZED is reflected to the partner program.

Scope of the Conversation_ID

In VM/ESA, only the application that allocates or accepts a conversation can use the associated conversation ID.

Identifying Product-Specific Errors

CPI Communications defines a return code called CM_PRODUCT_SPECIFIC_ERROR for each routine. In VM/ESA, when a call to a CPI Communications routine results in this return code:

- The file CPICOMM LOGDATA A is appended with a record describing the cause of the error. Each record is prefixed with "CMxxxx_PRODUCT_SPECIFIC_ERROR" when it is added to the CPICOMM LOGDATA A file. "CMxxxx" identifies the routine that produced the error. (While the CPICOMM LOGDATA file provides error information, it is not intended to be used as a programming interface.)
- The return code does not cause state changes.

This section lists the VM/ESA record entries possible with each CPI Communications routine's CM_PRODUCT_SPECIFIC_ERROR return code. Messages associated with that return code on VM/ESA extension routines are listed in the routine description.

Note: If a problem is encountered while attempting to write to the CPICOMM LOGDATA A file, the product-specific error message may not be written to the file although the application has received the CM_PRODUCT_SPECIFIC_ERROR return code.

The *code* in some messages is a decimal value representing:

- The 4-digit IPRCODE returned by an APPC/VM function.

  IPRCODEs are documented in the “Condition Codes and Return Codes” section of the specified function of the APPCVM macro, which is described in *VM/ESA: CP Programming Services*.

- The 4-digit return code given by a CMSIUCV or HNDIUCV function.

  The CMSIUCV or HNDIUCV return codes, which are padded on the left if less than 4 digits, are documented in the “Return Codes” section of those macros, both of which are described in the *VM/ESA: CMS Application Development Reference for Assembler*.

- The 5-digit CSL reason code returned by a CMSIUCV or HNDIUCV function.

  Reason codes are documented in the *VM/ESA: System Messages and Codes* book.

The *hexcode* in some messages is a hexadecimal value of 4 digits returned by a function of the APPCVM macro.

The VM/ESA-specific messages associated with each CPI Communications routine's CM_PRODUCT_SPECIFIC_ERROR return code are:

Accept_Conversation (CMACCP)

- CMSIUCV ACCEPT failed with CSL reason code *code*
- CMSIUCV ACCEPT failed with return code *code*
- HNDIUCV SET failed with return code *code*
- Unable to get storage
- Unable to RTNLOAD VMMTLIB.

Allocate (CMALLC)

- CMSIUCV CONNECT completed with code *code*
- APPCVM CONNECT completed by a sever interrupt with IPCODE *hexcode*
- The allocation cannot be to the application's own virtual machine
- Unable to set alternate user ID
- Privilege class not authorized to set alternate user ID
- Providing a security password without a security user ID is invalid.

Confirm (CMCFM)

- Unexpected IPRCODE *code* from APPCVM SENDCNF call.

Deallocate (CMDEAL)

- Unexpected IPRCODE *code* from APPCVM SENDCNF or SETMODFY call
- Error freeing storage for log data.

Flush (CMFLUS)

- Unexpected IPRCODE *code* from APPCVM SENDDATA call.

Initialize_Conversation (CMINIT)

- Bad Side-Information Security value
- Unable to get storage
- HNDIUCV SET failed with return code *code*
- Unable to RTNLOAD VMMTLIB.

Prepare_To_Receive (CMPTR)

- Unexpected IPRCODE *code* from APPCVM RECEIVE call.

Receive (CMRCV)

- APPCVM RECEIVE returned neither data nor status
- Unable to get storage
- Error freeing storage for log data
- Unexpected IPRCODE *code* from APPCVM SENDDATA or RECEIVE call.

Request_To_Send (CMRTS)

- Unexpected IPRCODE *code* from APPCVM SENDREQ call.

Send_Data (CMSEND)

- Unexpected IPRCODE *code* from APPCVM SENDDATA or RECEIVE call.

Send_Error (CMSERR)

- Unexpected IPRCODE *code* from APPCVM SENDERR call
- Error freeing storage for log data.

Set_Log_Data (CMSLD)

- Unable to get storage.

Set_Partner_LU_Name (CMSPLN)

- The partner LU name cannot contain a period
- A partner LU name field cannot contain more than 8 characters
- A blank in a partner LU name should only be used as a delimiter.

Diagnosing Errors

This section discusses log data processing, invocation errors, causes for selected return codes, and APPC protocol errors in VM/ESA.

Processing Log Data

If the *log_data* characteristic contains data (as a result of a Set_Log_Data call), log data is appended to a file called CPICOMM LOGDATA A under any of the following conditions:

- When the local program issues a Send_Error call
- When the local program issues a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND

- When the local program issues a Send_Data call with *send_type* set to CM_SEND_AND_DEALLOCATE and *deallocate_type* set to CM_DEALLOCATE_ABEND.

If the partner application is also using CPI Communications on a VM/ESA system, the log data is written to the partner's CPICOMM LOGDATA A file when one of the conditions above is reflected as a return code on a CPI Communications call to the partner application.

If the partner encounters a problem while receiving the log data, one of the following messages is written to the partner's CPICOMM LOGDATA A file:

- Log Data receive routine: Error getting storage.
- Log Data receive routine: Error freeing storage.
- Log Data receive routine: Unexpected IPRCODE *nnnn* from APPCVM RECEIVE call.

If an error is encountered while attempting to write to the CPICOMM LOGDATA A file, the log data will not be written to the file.

If the conversation for which the *log_data* characteristic was specified is a protected conversation (*sync_level* is set to CM_SYNC_POINT), the log data may be written during sync point processing to a file called CMSCOMM LOGDATA A, rather than to the CPICOMM LOGDATA A file.

Invocation Errors

If a VM/ESA program cannot successfully invoke the CPI Communications routine it is trying to call, the following error message is generated:

```
1292E Error calling CPI-Communications routine, return code
      retcode
```

If error message 1292E is received, the called routine was not invoked and the program was terminated with an abend code of X'ACB'. If the routine was called from REXX, no abend occurs and execution continues. The following return codes for this message are possible when using any of the supported languages:

Code	Meaning
-7	The CPI Communications routine called was not loaded. Issue the following command: `'RTNLOAD * (FROM VMLIB SYSTEM GROUP VMLIB)'` and then try calling the routine again. If this fails, contact the system administrator. (This command is ordinarily executed as part of the standard SYSPROF EXEC during IPL of the virtual machine.)
-8	The CPI Communications routine called has been dropped. Follow the same steps as for return code -7.
-9	Insufficient storage is available.
-10	Too many parameters were specified for the CPI Communications routine. Refer to the detailed description for the routine in this book to find the proper number of parameters.
-11	Not enough parameters were specified for the CPI Communications routine. Follow the same step as for return code -10.

The following return codes for this message are possible only when using REXX:

Code	Meaning
104	Insufficient virtual storage is available.
-3	Routine does not exist.
-20	A callable services library internal error occurred: invalid call.
-22	A callable services library internal error occurred: parameter list contains more than one argument.
-24	VM/REXX internal error: EXECCOMM FETCH failure.
-25	VM/REXX internal error: EXECCOMM SET failure.
-26*nnn*	Invalid data length for parameter number *nnn*.
-27*nnn*	Invalid data or data type for parameter number *nnn*.
-28*nnn*	Invalid variable name for parameter number *nnn*.
-29*nnn*	Invalid length value (for example, a negative value) was specified for length parameter, parameter number *nnn*.

(For the last four return codes, note that parameters are numbered serially, corresponding to the order in which they are coded. The routine name is always parameter number 001, the next parameter is 002, and so forth.)

See the *VM/ESA: System Messages and Codes* book for additional information.

Note: The CPI Communications interface is available only on VM/SP Release 6 and on VM/ESA. If a CPI Communications program compiled on VM/ESA Release 2 is run on a VM system that does not support CPI Communications, a return code of -12 is placed in register 15. In addition, the following invocation error message will be displayed:

```
CPI Communications not available on this release of CMS.  IPL
correct level of CMS.
```

Possible Causes for Selected Return Codes

Return codes for CPI Communications routines are listed with each routine in Chapter 4, and they are generically described in Appendix B; however, reasons for return codes may be specific to VM/ESA. The following list shows possible VM/ESA-specific causes for some CPI Communications return codes:

CM_RESOURCE_FAILURE_NO_RETRY
: This code can result when the partner does one of the following:

- Fails to deallocate the conversation before issuing the Terminate_Resource_Manager (XCTRRM) routine
- Re-IPLs CMS or logs off
- Issues an IUCV Sever
- Fails to deallocate the conversation before going through end-of-work unit or issuing the Return Workunitid (DMSRETWU) or Purge Workunitids (DMSPURWU) callable services library (CSL) routine to end the associated CMS work unit.

CM_SECURITY_NOT_VALID

This code can result for the following reasons:

- The user ID trying to allocate a conversation to a private resource is not authorized on the `:list.` tag in the private server's $SERVER$ NAMES file.
- Access security type on the conversation is NONE, but the remote program does not accept XC_SECURITY_NONE.
- An invalid password was supplied for a conversation with an access security type of XC_SECURITY_PROGRAM.
- An Identify_Resource_Manager (XCIDRM) call was issued with the *security_level_flag* set to XC_REJECT_SECURITY_NONE.

CM_TP_NOT_AVAILABLE_NO_RETRY

This code can result when the connection to the remote program cannot be completed due to one of the following:

- Either the local or the remote program does not have the appropriate IUCV authority in its VM directory authorization.
- The program tried allocating a conversation to a private resource manager program, but the private server virtual machine either had SET SERVER OFF or SET FULLSCREEN ON.
- The server virtual machine has exceeded its maximum number of connections.
- The private server virtual machine is running a TP-model application and has already accepted a protected conversation with the same LUWID as that associated with the protected conversation allocated by the local program. This situation is referred to as *allocation wrapback*.

CM_TPN_NOT_RECOGNIZED

This code can result when the connection to the remote program cannot be completed due to one of the following:

- The remote local, global, or system server virtual machine has not issued the Identify_Resource_Manager (XCIDRM) routine.
- No entry in the private server's $SERVER$ NAMES file matches the incoming resource ID (private resource name).
- The private server virtual machine cannot be autologged.
- The routine specified on the `:module.` tag in the private server's $SERVER$ NAMES file is unknown.
- There is no `:module.` tag for an entry in the private server's $SERVER$ NAMES file, and no program corresponds to the name specified on the `:nick.` tag.

APPC Protocol Errors in VM/ESA

It is possible, although unlikely, for CPI Communications to encounter an APPC architectural protocol error. If this condition arises in VM/ESA, a return code of either CM_RESOURCE_FAILURE_NO_RETRY or CM_RESOURCE_FAIL_NO_RETRY_BO is returned to the application, and a file called CPICOMM LOGDATA A is appended with a message line providing the error code that caused the protocol error.

This section lists the CPI Communications routines that can return a protocol error along with the associated message. The X'xxxx' value in the messages is one of:X'0410', X'0420',X'0510', X'0520', or X'0530'. These codes are documented in *VM/ESA: CP Programming Services*. Note that each message is prefixed with "CMxxxx_PROTOCOL_ERROR" when it is added to the CPICOMM LOGDATA A file. "CMxxxx" identifies the routine that produced the error. If a problem is encountered while attempting to write to the CPICOMM LOGDATA A file, the protocol error message will not be written to the file.

Confirm (CMCFM)

- APPCVM SENDCNF completed with IPCODE X'xxxx' and IPWHATRC X'03'.

Deallocate (CMDEAL)

- APPCVM SEVER completed with IPCODE X'xxxx' and IPWHATRC X'03'.

Prepare_To_Receive (CMPTR)

- APPCVM SENDCNF completed with IPCODE X'xxxx' and IPWHATRC X'03'.

Receive (CMRCV)

- APPCVM RECEIVE completed with IPCODE X'xxxx' and IPWHATRC X'03'.
- A mapped conversation GDS variable with GDSID X'12F2' has been received.

Send_Data (CMSEND)

- APPCVM SENDDATA completed with IPCODE X'xxxx' and IPWHATRC X'03'.

Send_Error (CMSERR)

- APPCVM SENDERR completed with IPCODE X'xxxx' and IPWHATRC X'03'.

When Allocation Requests Are Sent

If the target program is within the same TSAF or CS collection, Allocate (CMALLC) provides no session buffering; the Allocate call can be considered to include the function of the Flush (CMFLUS) call. Therefore, the allocation request is flowed to the partner immediately. When the conversation crosses the SNA network, the allocation request is buffered and is sent when the buffer is flushed.

Deviations from the CPI Communications Architecture

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

- For the Initialize_Conversation call, if the value specified for the *sym_dest_name* does not match an entry in side information VM/ESA does not return CM_PROGRAM_PARAMETER_CHECK. Instead, CM_OK is returned and the specified *sym_dest_name* is used to set the TP_name characteristic for the conversation.
- CPI Communications calls from RPG programs are not currently supported on VM/ESA systems.

- On VM/ESA systems, when a conversation crosses a VTAM network, the Receive call does not return data until an entire logical record arrives at the local system. For basic conversations, this behavior may cause the Receive call to return unexpected results.

  For example, a Receive call with *receive_type*=CM_RECEIVE_IMMEDIATE will return a return code of CM_UNSUCCESSFUL if the entire logical record has not arrived at the local system, even if enough of the logical record has arrived to satisfy the Receive call's requested length. Similarly, a Receive call with *receive_type*=CM_RECEIVE_AND_WAIT waits until the remainder of the logical record is received by the local system, even if enough of the logical record has arrived to satisfy the requested length. This call works properly in conversations within a TSAF or CS collection.
- On VM/ESA systems, for conversations that cross a VTAM network, the Test_Request_To_Send_Received (CMTRTS) call always sets *request_to_send_received* to CM_REQ_TO_SEND_NOT_RECEIVED when *return_code*=CM_OK, regardless of whether the remote programs have sent such requests to the local programs. This call works properly under VM/ESA within a TSAF or CS collection.
- On VM/ESA, a space is used as a delimiter instead of a period in the *partner_LU_name*.
- On VM/ESA, CPI Communications declares an APPC protocol error when a user-control data GDS variable is received.
- On VM/ESA, any program initialization parameter variables received by CPI Communications are ignored.

VM/ESA Extension Calls

Table 44 on page 542 summarizes VM/ESA routines that are extensions to CPI Communications. The routines are listed in alphabetic order by their callable name. The last column of the table shows the page where the routine is described in detail.

These routines can be used in CMS to take advantage of VM/ESA's capabilities. Note, however, that because these extension routines may not be supported or may be implemented differently in other SAA operating environments, a program using any of these VM/ESA extension routines cannot be moved to another system without being changed.

Notes:

1. To aid in recognizing the call names, all VM/ESA extension routines begin with the prefix **XC**.
2. See “VM/ESA Variables and Characteristics” on page 576 for information on the possible values for the variables and characteristics associated with the VM/ESA extension routines.

Table 44 (Page 1 of 2). Overview of VM/ESA Extension Routines

Call	Pseudonym	Description	Page
XCECL	Extract_Conversation_LUWID	Lets a program extract the SNA LU 6.2 architected logical unit of work identifier for a given protected conversation.	544
XCECSU	Extract_Conversation_Security_User_ID	Lets a program extract the access security user ID associated with a given conversation.	546
XCECWU	Extract_Conversation_Workunitid	Lets a program extract the CMS work unit ID for a given conversation.	548
XCELFQ	Extract_Local_Fully_Qualified_LU_Name	Lets a program extract the local fully-qualified LU name for a given conversation.	550
XCERFQ	Extract_Remote_Fully_Qualified_LU_Name	Lets a program extract the remote fully-qualified LU name for a given conversation.	552
XCETPN	Extract_TP_Name	Lets a program extract the TP name for a given conversation.	554
XCIDRM	Identify_Resource_Manager	Declares to CMS a name (resource ID) by which the resource manager application will be known.	555
XCSCSP	Set_Conversation_Security_Password	Sets the access security password value for the conversation. The target LU uses this value and the security user ID to verify the identity of the requester.	562
XCSCST	Set_Conversation_Security_Type	Sets the access security type for the conversation. The security type determines what security information is sent to the target.	564
XCSCSU	Set_Conversation_Security_User_ID	Sets the access security user ID value for the conversation. The target LU uses this value and the access security password to verify the identity of the requester.	566
XCSCUI	Set_Client_Security_User_ID	Lets an intermediate server specify an alternate user ID (the user ID of a specific client application).	559
XCSUE	Signal_User_Event	Queues an event to be reported by a subsequent call to Wait_on_Event (XCWOE) in the virtual machine.	568

Table 44 (Page 2 of 2). Overview of VM/ESA Extension Routines

Call	Pseudonym	Description	Page
XCTRRM	Terminate_Resource_Manager	Ends ownership of a resource by a resource manager program.	570
XCWOE	Wait_on_Event	Allows an application to wait on communications from one or more partners. Events posted are user events, allocation requests, information input, notification that resource management has been revoked, console input, and asynchronous Shared File System (SFS) requests.	571

Extract_Conversation_LUWID (XCECL)

A program uses the Extract_Conversation_LUWID (XCECL) call to extract the SNA LU 6.2 architected logical unit of work ID (LUWID) for a given protected conversation. The LUWID can be used to identify the most recent sync point.

This routine can be called after issuing an Allocate (CMALLC) or Accept_Conversation (CMACCP) call to establish a protected conversation.

Note: The Extract_Conversation_LUWID call is valid only for protected (*sync_level*=CM_SYNC_POINT) conversations.

Format

```
CALL XCECL(conversation_ID,
           luwid,
           luwid_length,
           return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

luwid *(output)*
Is a 26-byte character variable used to return the SNA LU 6.2 architected LUWID associated with the specified conversation ID when the *return_code* is CM_OK. See the usage note for a description of the LUWID.

luwid_length *(output)*
Is a variable used to return the length of the SNA LU 6.2 architected LUWID when the *return_code* is CM_OK.

return_code *(output)*
Is a variable used to hold the return code passed back from the communications routine to the calling program. The *return_code* variable can have one of the following values:

- CM_OK
  Successful completion.
- CM_PRODUCT_SPECIFIC_ERROR
  This return code can result in the CPICOMM LOGDATA A file being appended with one of the following entries:

  ```
  XCECL_PRODUCT_SPECIFIC_ERROR:  Call to DMSLUWID failed

  XCECL_PRODUCT_SPECIFIC_ERROR:  Call to CMSIUCV QCMSWID failed
  ```
- CM_PROGRAM_PARAMETER_CHECK
  This can result from one of the following conditions:
  – The *conversation_ID* specifies an unassigned conversation identifier.
  – The *conversation_ID* was not for a protected (*sync_level*=CM_SYNC_POINT) conversation.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is in **Initialize** state.
- CM_OPERATION_NOT_ACCEPTED
  This value indicates that a previous operation on this conversation is incomplete.

State Changes

This call does not cause a state change.

Usage Notes

1. The LUWID can be up to 26 bytes long, containing the following fields:

Length	Description
1	Length of the fully-qualified LU name
1-17	The fully-qualified LU name. If its length is less than 17 bytes, it is left-justified and padded on the right with blanks (X'40'). It is composed of the following fields:
6	Instance number in binary
2	Sequence number in binary

   Fields of the fully-qualified LU name:

Length	Description
0-8	The network ID
0-1	A delimiter (a period)
1-8	LU name

   If both the network ID and the LU name are present, they are separated by a period.

   Refer to the *SNA Format and Protocol Reference Manual for LU Type 6.2* for more information about the LUWID.

2. This call does not change the *luwid* for the specified conversation.

Extract_Conversation_Security_User_ID (XCECSU)

A program uses the Extract_Conversation_Security_User_ID (XCECSU) routine to extract the access security user ID associated with a given conversation.

A security user ID is only returned if *conversation_security_type* is XC_SECURITY_SAME or XC_SECURITY_PROGRAM. If the *conversation_security_type* is XC_SECURITY_NONE, the *security_user_ID* parameter returns nulls (X'00'), and the length is set to zero.

The returned *security_user_ID* can be used as input to the DMSREG (Resource Adapter Registration) CSL routine, which is described in the *VM/ESA: CMS Application Development Reference*. The *security_user_ID* returned is not valid for this use if Extract_Conversation_Security_User_ID is called while the conversation is in **Initialize** state.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

security_user_ID *(output)*
Is a variable used to return the access security user ID obtained by this routine when the *return_code* is CM_OK.

security_user_ID_length *(output)*
Is a variable used to return the length, in bytes, of the *security_user_ID* when the *return_code* is CM_OK.

return_code *(output)*
Is a variable used to hold the return code passed back from the communications routine to the calling program. The *return_code* variable can have one of the following values:

- CM_OK
  Successful completion.
- CM_PROGRAM_PARAMETER_CHECK
  The *conversation_ID* specifies an unassigned conversation identifier.
- CM_OPERATION_NOT_ACCEPTED
  This value indicates that a previous operation on this conversation is incomplete.

State Changes

This call does not cause a state change.

Usage Notes

1. Generally, when in **Initialize** state the *security_user_ID_length* is zero and the *security_user_ID* is meaningless. If the conversation was initialized with the `:security.` tag set to PGM in side information (a CMS communications directory file) and a security user ID was specified on the `:userid.` tag, then that specified value is returned. Also, if Set_Conversation_Security_User_ID (XCSCSU) was called to set a security user ID, that value is returned.

   When Extract_Conversation_Security_User_ID is called after a successful Allocate (CMALLC) or Accept_Conversation (CMACCP) call, the access security user ID for the specified conversation ID is returned.

2. This call does not change the *security_user_ID* for the specified conversation.

Extract_Conversation_Workunitid (XCECWU)

A program uses the Extract_Conversation_Workunitid (XCECWU) call to extract the CMS work unit ID for a given conversation. This routine can be used after issuing an Allocate (CMALLC) or Accept_Conversation (CMACCP) call. This routine 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 DMSPUSWU (Push Workunitid) for changing the default CMS work unit, and DMSCOMM (Commit) and DMSROLLB (Rollback) when using Coordinated Resource Recovery. These callable services library (CSL) routines are described in the *VM/ESA: CMS Application Development Reference*.

For information on CMS work units, refer to the *VM/ESA: CMS Application Development Guide*.

Format

```
CALL XCECWU(conversation_ID,
            workunitid,
            return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

workunitid *(output)*
Is a signed 4-byte integer variable used to return the CMS work unit ID associated with the specified conversation ID when the *return_code* is CM_OK.

return_code *(output)*
Is a variable used to hold the return code passed back from the communications routine to the calling program. The *return_code* variable can have one of the following values:

- CM_OK
  Successful completion.
- CM_PRODUCT_SPECIFIC_ERROR
  When this code is returned, a file named CPICOMM LOGDATA A is appended with the following line:

  ```
  XCECWU_PRODUCT_SPECIFIC_ERROR:  Call to CMSIUCV QCMSWID failed
  ```

- CM_PROGRAM_PARAMETER_CHECK
  The *conversation_ID* specifies an unassigned conversation identifier.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is in **Initialize** state.
- CM_OPERATION_NOT_ACCEPTED
  This value indicates that a previous operation on this conversation is incomplete.

State Changes

This call does not cause a state change.

Usage Notes

This call does not change the *workunitid* for the specified conversation.

Extract_Local_Fully_Qualified_LU_Name (XCELFQ)

A program uses the Extract_Local_Fully_Qualified_LU_Name (XCELFQ) call to extract the local fully-qualified LU name for a given conversation. This routine can be used after issuing an Allocate (CMALLC) or Accept_Conversation (CMACCP) call.

The output from this routine can be used as input on the DMSREG (Resource Adapter Registration) CSL routine, which is described in the *VM/ESA: CMS Application Development Reference*.

Format

```
CALL XCELFQ(conversation_ID,
            local_FQ_LU_name,
            local_FQ_LU_name_length,
            return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

local_FQ_LU_name *(output)*
Specifies the variable used to return the local fully-qualified LU name obtained by this routine when the *return_code* is CM_OK. Allow 17 bytes for this variable: 0 to 8 bytes for a network ID, 1 to 8 bytes for the LU name, and 1 byte for a delimiter if both a network ID and LU name are specified (the delimiter is a period). If the fully-qualified LU name is less than 17 bytes long, it is left-justified and padded on the right with blanks (X'40').

local_FQ_LU_name_length *(output)*
Specifies the variable used to return the length of the local fully-qualified LU name obtained by this routine when the *return_code* is CM_OK. This length is zero if the partner program is on the same LU as the program issuing this routine (communication is not routed through AVS).

return_code *(output)*
Is a variable used to hold the return code passed back from the communications routine to the calling program. The *return_code* variable can have one of the following values:

- CM_OK
  Successful completion.
- CM_PROGRAM_PARAMETER_CHECK
  The *conversation_ID* specifies an unassigned conversation identifier.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is in **Initialize** state.
- CM_OPERATION_NOT_ACCEPTED
  This value indicates that a previous operation on this conversation is incomplete.

State Changes

This call does not cause a state change.

Usage Notes

This call does not change the *local_FQ_LU_name* for the specified conversation.

Extract_Remote_Fully_Qualified_LU_Name (XCERFQ)

A program uses the Extract_Remote_Fully_Qualified_LU_Name (XCERFQ) call to extract the remote fully-qualified LU name for a given conversation. This routine can be used after issuing an Allocate (CMALLC) or Accept_Conversation (CMACCP) call.

The output from this routine can be used as input on the DMSREG (Resource Adapter Registration) CSL routine, which is described in the *VM/ESA: CMS Application Development Reference*.

Format

```
CALL XCERFQ(conversation_ID,
            remote_FQ_LU_name,
            remote_FQ_LU_name_length,
            return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

remote_FQ_LU_name *(output)*
Specifies the variable to contain the remote fully-qualified LU name obtained by this routine and returned to the local program when the *return_code* is CM_OK. Allow 17 bytes for this variable: 0 to 8 bytes for a network ID, 1 to 8 bytes for the LU name, and 1 byte for a delimiter if both a network ID and LU name are specified (the delimiter is a period). If the fully-qualified LU name is less than 17 bytes long, it is left-justified and padded on the right with blanks (X'40').

remote_FQ_LU_name_length *(output)*
Specifies the variable to contain the length of the remote fully-qualified LU name obtained by this routine and returned to the local program when the *return_code* is CM_OK. This length is zero if the remote program is on the same system as the local program.

return_code *(output)*
Is a variable used to hold the return code passed back from the communications routine to the calling program. The *return_code* variable can have one of the following values:

- CM_OK
  Successful completion.
- CM_PROGRAM_PARAMETER_CHECK
  The *conversation_ID* specifies an unassigned conversation identifier.
- CM_PROGRAM_STATE_CHECK
  This value indicates that the conversation is in **Initialize** state.
- CM_OPERATION_NOT_ACCEPTED
  This value indicates that a previous operation on this conversation is incomplete.

State Changes

This call does not cause a state change.

Usage Notes

This call does not change the *remote_FQ_LU_name* for the specified conversation.

Extract_TP_Name (XCETPN)

A program uses the Extract_TP_Name (XCETPN) call to extract the *TP_name* characteristic for a given conversation.

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

TP_name *(output)*
Specifies the variable to contain the transaction program name obtained by this routine and returned to the local program when the *return_code* is CM_OK. Allow 64 bytes for this variable.

TP_name_length *(output)*
Specifies the variable to contain the length of the transaction program name obtained by this routine and returned to the local program when the *return_code* is CM_OK.

return_code *(output)*
Specifies the return code that is passed back from the communications routine to the calling program. The *return_code* variable can have one of the following values:

- CM_OK
  Successful completion.
- CM_PROGRAM_PARAMETER_CHECK
  The *conversation_ID* specifies an unassigned conversation identifier.
- CM_OPERATION_NOT_ACCEPTED
  This value indicates that a previous operation on this conversation is incomplete.

State Changes

This call does not cause a state change.

Usage Notes

1. This call does not change the *TP_name* for the specified conversation.
2. When the non-blank symbolic destination name is specified on the Initialize_Conversation (CMINIT) call, the *TP_name* characteristic is set to an initial value from side information if an entry exists for it (on the `:tpn.` tag). If there is no entry in side information or a different value is required, the TP name can be set using the Set_TP_Name (CMSTPN) call.

Identify_Resource_Manager (XCIDRM)

An application uses the Identify_Resource_Manager (XCIDRM) routine to declare to CMS the name of a resource that it wants to manage.

Format

```
CALL XCIDRM(resource_ID,
            resource_manager_type,
            service_mode,
            security_level_flag,
            return_code)
```

Parameters

resource_ID *(input)*
: Specifies the name of a resource managed by this resource manager application. This variable must be an 8-byte character string, padded on the right with blanks if necessary. The first character of the resource name must be alphanumeric.

  The value of the *resource_ID* parameter corresponds to the *TP_name* characteristic that requesting applications supply when allocating a conversation to this resource. This *resource_ID* name must be unique within the scope (private, local, global, or system) for which it is identified.

  Other programs' allocation requests are then routed to the application that called this Identify_Resource_Manager routine.

resource_manager_type *(input)*
: Identifies whether the specified *resource_ID* is a private, local, global, or system resource. The *resource_manager_type* variable can have one of the following values:

  - XC_PRIVATE
    Private resource names are identified only to the virtual machine in which they are active, but they can be accessed by authorized users on the same VM/ESA system, in the same TSAF or CS collection, or on another system in an SNA network.
  - XC_LOCAL
    Local resource names are identified only to the system in which they reside and cannot be accessed from outside this system.
  - XC_GLOBAL
    Global resource names are identified to an entire TSAF or CS collection. They may be accessed by other users in the collection or in an SNA network.
  - XC_SYSTEM
    System resource names are identified only to the VM/ESA system in which they reside, but are remotely accessible from other systems.

service_mode *(input)*

Indicates how this resource manager application handles conversations associated with the specified *resource_ID*. The *service_mode* can have one of the following values:

- XC_SINGLE
  This resource manager program can accept only a single conversation for the specified *resource_ID*.

  If the resource manager program already has accepted a conversation for the resource and another program requests that same resource, the identification of the resource manager type as private, local, global, or system determines what happens to the new allocation request:

  – For a private resource, CMS queues the new allocation request. Then, when the private resource manager program that is running ends, CMS automatically restarts the private resource manager program and takes the first pending allocation request off the queue.

  – For a local, global, or system resource, CMS deallocates the allocation request. The application issuing the allocation request receives a *return_code* value of CM_TPN_NOT_RECOGNIZED upon completion of a subsequent call that allows this return code.

- XC_SEQUENTIAL
  This resource manager program can accept only one conversation at a time for the specified *resource_ID*. When one conversation is completed and deallocated, the resource manager program can issue Wait_on_Event (XCWOE) to wait for the next allocation request, or issue Accept_Conversation (CMACCP). (If a program issues Accept_Conversation and there is not a pending allocation request, however, a CM_PROGRAM_STATE_CHECK return code is returned.)

  If the resource manager program has an active conversation for the resource and another program requests that same resource, the identification of the resource manager type as private, local, global, or system determines what happens to the new allocation request:

  – For a private resource, CMS queues the new allocation request. Then after the active conversation is deallocated, the resource manager program should issue Wait_on_Event or Accept_Conversation as described in the preceding paragraph.

  – For a local, global, or system resource, CMS deallocates the allocation request. The application issuing the allocation request receives a *return_code* value of CM_TPN_NOT_RECOGNIZED upon completion of a subsequent call that allows this return code.

- XC_MULTIPLE
  This resource manager program can accept multiple conversations for the specified *resource_ID*.

  If another program requests the resource for which there is already an active conversation, the pending allocation request will be reported to the resource manager on a future call to Wait_on_Event.

security_level_flag *(input)*

Indicates whether this resource manager will accept inbound connections for the specified *resource_ID* that have *conversation_security_type* equal to XC_SECURITY_NONE. The *security_level_flag* variable must have one of the following values:

- XC_REJECT_SECURITY_NONE
  The resource manager will not accept connections that have a *conversation_security_type* of XC_SECURITY_NONE. A requester program that allocates a conversation with XC_SECURITY_NONE will get a return code of CM_SECURITY_NOT_VALID upon completion of a subsequent call that allows this return code.
- XC_ACCEPT_SECURITY_NONE
  The resource manager will accept connections that have *conversation_security_type* equal to XC_SECURITY_NONE.

return_code *(output)*

Is a variable used to hold the return code passed back from the communications routine to the calling program. The *return_code* variable can have one of the following values:

- CM_OK
  Successful completion.
- CM_PRODUCT_SPECIFIC_ERROR
  This return code can result from one of the following conditions:
  - A storage failure prevented the specified *resource_ID* from being identified. When this code is returned, a file named CPICOMM LOGDATA A is appended with the following line:

    ```
    XCIDRM_PRODUCT_SPECIFIC_ERROR:  Unable to get storage
    ```

  - An error was encountered trying to load the VMMTLIB callable services library which is required by CPI Communications. When this code is returned, a file named CPICOMM LOGDATA A is appended with the following line:

    ```
    XCIDRM_PRODUCT_SPECIFIC_ERROR:  Unable to RTNLOAD VMMTLIB
    ```

- CM_PROGRAM_PARAMETER_CHECK
  This can result from one of the following conditions:
  - The *resource_ID* has already been defined within the virtual machine.
  - The *resource_manager_type* contains an invalid value.
  - The *service_mode* contains an invalid value.
  - The *security_level_flag* contains an invalid value.
- CM_PROGRAM_STATE_CHECK
  This is a *TP-model application*, so this routine cannot be called.
- CM_UNSUCCESSFUL
  Identify_Resource_Manager was unable to obtain ownership of the resource. The following are possible reasons:
  - The resource is already owned by another virtual machine.
  - The virtual machine in which the application is running does not have authority to connect to *IDENT.
  - The virtual machine in which the application is running does not have authority to declare the resource.

  This return code applies only when *resource_manager_type* is XC_LOCAL, XC_GLOBAL, or XC_SYSTEM.

State Changes

This call is not specific to a conversation, so it does not cause a state change.

Usage Notes

1. An application can call Identify_Resource_Manager multiple times to identify different resources that it wants to manage.
2. The application does not need to call Identify_Resource_Manager if:
   - The application initiates all its conversations and is never the target of an allocation request.
   - The application is a private resource manager invoked by CMS as the target of a single conversation.
3. An application calling Identify_Resource_Manager should also call the Terminate_Resource_Manager (XCTRRM) routine before exiting. See "End-of-Command Processing" on page 578 for information regarding system cleanup of resource identifiers after program termination.

Set_Client_Security_User_ID (XCSCUI)

A program acting as an intermediate server uses the Set_Client_Security_User_ID (XCSCUI) routine to set the access security user ID 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.

An intermediate server can call Set_Client_Security_User_ID 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* 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 is privilege class B (unless default privilege classes have been changed). If not authorized, the Allocate (CMALLC) call will complete with a CM_PRODUCT_SPECIFIC_ERROR return code.
- There is no alternate user ID in effect for an incoming conversation. If an alternate user ID is in effect, the Allocate will complete with a CM_PRODUCT_SPECIFIC_ERROR return code.

Note: Set_Client_Security_User_ID can be issued only for a conversation that is in **Initialize** state. It cannot be issued after an Allocate.

Format

```
CALL XCSCUI(conversation_ID,
            client_user_ID,
            return_code)
```

Parameters

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

client_user_ID *(input)*
: Specifies the client's user ID. This variable must be an 8-byte character string, padded on the right with blanks as necessary.

return_code *(output)*
Is a variable used to hold the return code passed back from the communications routine to the calling program. The *return_code* variable can return one of the following values:

- CM_OK
  Successful completion.
- CM_PROGRAM_PARAMETER_CHECK
  This can result from one of the following conditions:
  – The *conversation_ID* specifies an unassigned conversation identifier.
  – The *conversation_security_type* for the specified conversation is not equal to XC_SECURITY_SAME.
- CM_PROGRAM_STATE_CHECK
  This can result from one of the following conditions:
  – The conversation is not in **Initialize** state.
  – This is a *TP-model application*, so this routine cannot be called.
- CM_OPERATION_NOT_ACCEPTED
  This value indicates that a previous operation on this conversation is incomplete.

State Changes

This call does not cause a state change.

Usage Notes

1. A program that acts as an intermediate server might have incoming conversations from various clients. With Set_Client_Security_User_ID, such a server can specify a particular user ID that will be presented to a target resource manager. In this way, the target resource manager can determine whether to grant access to its resources based on the client's access security user ID rather than the intermediate server's.

2. Here is a typical sequence of events that include an intermediate server calling the Set_Client_Security_User_ID:

   a. The server application issues Identify_Resource_Manager (XCIDRM) to declare the resource it is managing, and then issues Wait_on_Event (XCWOE) to wait for a request.

   b. A client application issues an Allocate for a conversation to the server application.

   c. The server application accepts the conversation using the Accept_Conversation (CMACCP) routine.

   d. The server application calls the Extract_Conversation_Security_User_ID routine for the conversation with the client application to get the client’s access security user ID.

   e. The server calls Initialize_Conversation (CMINIT) to get a conversation ready to allocate on behalf of the client. The *conversation_security_type* characteristic should be set to XC_SECURITY_SAME, which is the default value.

   f. The server application calls Set_Client_Security_User_ID using the extracted access security user ID to set the security information of the new conversation.

g. The intermediate server application calls Allocate for the outgoing conversation on behalf of the client application.

h. The target program is presented with the client program's access security user ID.

3. When a TP-model intermediate server allocates a conversation to a target with *conversation_security_type* set to XC_SECURITY_SAME, the access security user ID forwarded on the allocation is always the client's user ID; TP-model intermediate servers cannot use Set_Client_Security_User_ID.
4. When an intermediate server that has called Identify_Resource_Manager allocates a conversation with *conversation_security_type* set to XC_SECURITY_SAME to the target resource manager without first calling Set_Client_Security_User_ID, the access security user ID forwarded on the allocation is its own user ID, not that of the client.
5. If a *return_code* other than CM_OK is returned on the call, the *client_user_ID* conversation characteristic is unchanged.

Set_Conversation_Security_Password (XCSCSP)

A program uses the Set_Conversation_Security_Password (XCSCSP) routine to set the access security password for a given conversation when the *conversation_security_type* is XC_SECURITY_PROGRAM. Both an access security user ID and password are required to establish a conversation with a security type of PGM.

This call does not change the value of the `:password.` tag in side information. Set_Conversation_Security_Password changes the *security_password* and *security_password_length* only for this conversation.

Note: Set_Conversation_Security_Password can be issued only for a conversation that is in **Initialize** state. It cannot be issued after an Allocate (CMALLC).

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

security_password *(input)*
Specifies the access security password. The target LU verifies the access security user ID and password pair during the allocation process. CPI Communications maintains this password while the conversation is in **Initialize** state.

security_password_length *(input)*
Specifies the length of the security password. The length must be specified as a value from 0 to 8. If 0 is specified, the password is set to null and the *security_password* parameter is ignored.

return_code *(output)*
Is a variable used to hold the return code passed back from the communications routine to the calling program. The *return_code* variable can return one of the following values:

- CM_OK
  Successful completion.
- CM_PROGRAM_PARAMETER_CHECK
  This can result from one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *security_password_length* variable specifies a value of less than 0 or greater than 8.
- CM_PROGRAM_STATE_CHECK
  This can result from one of the following:
  - The conversation is not in **Initialize** state.
  - The *conversation_security_type* is not XC_SECURITY_PROGRAM.
- CM_OPERATION_NOT_ACCEPTED
  This value indicates that a previous operation on this conversation is incomplete.

State Changes

This call does not cause a state change.

Usage Notes

1. A program does not need to call this routine if a security password is specified in either of the following:

   - Side information (communications directory file)
   - An APPCPASS statement in the virtual machine's CP directory.

   The access security password specified on this routine overrides a password in the communications directory file and takes precedence over a password specified on a corresponding APPCPASS directory statement. If the *security_password_length* parameter is specified as zero, however, the CP directory will still be checked during conversation allocation for an APPCPASS statement to supply the missing password. The Set_Conversation_Security_User_ID routine works the same way.

2. If a *return_code* other than CM_OK is returned on the call, the *security_password* and *security_password_length* conversation characteristics are unchanged.

Set_Conversation_Security_Type (XCSCST)

A program uses the Set_Conversation_Security_Type (XCSCST) routine to set the security type for a given conversation. This routine overrides the value that was assigned when the conversation was initialized.

This call does not change the value of the `:security.` tag in side information. Set_Conversation_Security_Type changes the *conversation_security_type* only for this conversation.

Note: Set_Conversation_Security_Type can be issued only for a conversation that is in **Initialize** state. It cannot be issued after an Allocate (CMALLC).

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

conversation_security_type *(input)*
Specifies the kind of access security information to be sent to the target. The target LU uses this security information to verify the identity of the source. The access security information, if present, consists of either a user ID or a user ID and password. This parameter must be set to one of the following values:

- XC_SECURITY_NONE
  No access security information is to be included on the allocation request to the target resource manager.
- XC_SECURITY_SAME
  An access security user ID is sent on the allocation request to the target resource manager.
- XC_SECURITY_PROGRAM
  An access security user ID and password must be supplied. These values can be set by calls to Set_Conversation_Security_User_ID (XCSCSU) and Set_Conversation_Security_Password (XCSCSP) or from information contained in an APPCPASS statement in the virtual machine's CP directory.

return_code *(output)*
Is a variable used to hold the return code passed back from the communications routine to the calling program. The *return_code* variable can return one of the following values:

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

- CM_OPERATION_NOT_ACCEPTED
  This value indicates that a previous operation on this conversation is incomplete.

State Changes

This call does not cause a state change.

Usage Notes

1. A program does not need to use this routine if the default security type of XC_SECURITY_SAME is desired, or if a security type is specified in the virtual machine's side information.
2. If the current security type for the conversation is PGM, a security user ID previously set by a `:userid.` entry in side information associated with this conversation or by a call to Set_Conversation_Security_User_ID (XCSCSU) is reset to a null value if Set_Conversation_Security_Type is called to change the *conversation_security_type* to either XC_SECURITY_NONE or XC_SECURITY_SAME. Similarly, a security password previously set from a `:password.` tag in side information or by a Set_Conversation_Security_Password (XCSCSP) call is reset to a null value if the *conversation_security_type* is changed to either XC_SECURITY_NONE or XC_SECURITY_SAME.

   If the current security type for the conversation is SAME, a client security user ID previously set by a Set_Client_Security_User_ID (XCSCUI) call is reset to a null value if Set_Conversation_Security_Type is called to change the *conversation_security_type* to either XC_SECURITY_NONE or XC_SECURITY_PROGRAM.
3. If a *return_code* other than CM_OK is returned on the call, the *conversation_security_type* conversation characteristic is unchanged.

Set_Conversation_Security_User_ID (XCSCSU)

A program uses the Set_Conversation_Security_User_ID (XCSCSU) routine to set the access security user ID for a given conversation when the *conversation_security_type* is XC_SECURITY_PROGRAM. Both an access security user ID and password are required to establish a conversation when the *conversation_security_type* is XC_SECURITY_PROGRAM.

This call does not change the value of the `:userid.` tag in the side information. Set_Conversation_Security_User_ID changes the *security_user_ID* and *security_user_ID_length* only for this conversation.

Note: Set_Conversation_Security_User_ID can be issued only for a conversation that is in **Initialize** state. It cannot be issued after an Allocate (CMALLC).

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

security_user_ID *(input)*
Specifies the user ID. The target LU verifies the access security user ID and password, which can be specified on the Set_Conversation_Security_Password (XCSCSP) routine, during the allocation process. In addition, the target LU can use the user ID for auditing or accounting purposes.

security_user_ID_length *(input)*
Specifies the length, in bytes, of the security user ID. This variable must be given a value from 0 to 8. If 0 is specified, the user ID is set to null and the *security_user_ID* parameter is ignored.

return_code *(output)*
Is a variable used to hold the return code passed back from the communications routine to the calling program. The *return_code* variable can return one of the following values:

- CM_OK
  Successful completion.
- CM_PROGRAM_PARAMETER_CHECK
  This can result from one of the following:
  - The *conversation_ID* specifies an unassigned conversation identifier.
  - The *security_user_ID_length* variable specifies a value less than 0 or greater than 8.
- CM_PROGRAM_STATE_CHECK
  This can result from one of the following:
  - The conversation is not in **Initialize** state.
  - The *conversation_security_type* is not XC_SECURITY_PROGRAM.

- CM_OPERATION_NOT_ACCEPTED
  This value indicates that a previous operation on this conversation is incomplete.

State Changes

This call does not cause a state change.

Usage Notes

1. A program does not need to call this routine if a security user ID is specified in either of the following:
   - Side information (a communications directory file)
   - An APPCPASS statement in the virtual machine's CP directory.

   The access security user ID specified on this routine overrides a user ID in the communications directory file and takes precedence over a user ID specified on a corresponding APPCPASS directory statement. If the *security_user_ID_length* parameter is specified as zero, however, the CP directory will still be checked during conversation allocation for an APPCPASS statement to supply the missing user ID. The Set_Conversation_Security_Password routine works the same way.
2. If a *return_code* other than CM_OK is returned on the call, the *security_user_ID* and *security_user_ID_length* conversation characteristics are unchanged.

Signal_User_Event (XCSUE)

A program, such as an interrupt handler, uses the Signal_User_Event (XCSUE) routine to inform VM/ESA CPI Communications that a user event has occurred.

Signal_User_Event would typically be called from an interrupt 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. Information on writing interrupt handlers can be found in the *VM/ESA: CMS Application Development Guide for Assembler*. An example program using the Signal_User_Event call can be found in the *VM/ESA: CMS Application Development Guide*. The CPI Communications program must issue Wait_on_Event (XCWOE) to get the user-event notification.

Format

```
CALL XCSUE(event_ID,
           user_data,
           user_data_length,
           return_code)
```

Parameters

event_ID *(input)*
Specifies a variable to identify the event. This identifier will be returned by Wait_on_Event (XCWOE) in the *resource_ID* parameter when the event is reported.

user_data *(input)*
Specifies information about the event. The data supplied for this parameter is any information the program wants to supply to describe the event. This information will be returned by Wait_on_Event in the *event_buffer* parameter when the event is reported.

user_data_length *(input)*
Specifies the length of the information specified for the *user_data* parameter. If zero is specified, the *user_data* parameter is ignored. The length can be from 0 to 130 bytes.

return_code *(output)*
Is a variable used to hold the return code passed back from the communications call to the calling program. The *return_code* variable can have one of the following values:

- CM_OK
  Successful completion.
- CM_PROGRAM_PARAMETER_CHECK
  The *user_data_length* parameter specifies a value of less than 0 or greater than 130.
- CM_PRODUCT_SPECIFIC_ERROR
  A storage failure prevented the user event from being queued. A file named CPICOMM LOGDATA A is appended with the following line:

  ```
  XCSUE_PRODUCT_SPECIFIC_ERROR: Unable to get storage
  ```

- CM_PROGRAM_STATE_CHECK
  This value indicates that a successful call has not been made to Accept_Conversation (CMACCP), Initialize_Conversation (CMINIT), or Identify_Resource_Manager (XCIDRM).

State Changes

This call does not cause a state change.

Usage Notes

1. When Signal_User_Event is issued, VM/ESA CPI Communications queues the event so it can be reported by a subsequent Wait_on_Event call. Any number of events can be queued, limited only by the amount of storage in the virtual machine. If there is an outstanding Wait_on_Event call when Signal_User_Event is issued, Wait_on_Event reports the user event as soon as the interrupt handler completes.
2. Pending (unreported) user events are purged at end-of-command.
3. By using CMS event management services, events can be created and signalled without issuing the Signal_User_Event call.

Terminate_Resource_Manager (XCTRRM)

A resource manager application uses the Terminate_Resource_Manager (XCTRRM) routine to end management of a resource. CPI Communications automatically deallocates all conversations and pending allocation requests for the specified resource ID. All CPI Communications calls issued on these conversations following the Terminate_Resource_Manager call will result in a CM_PROGRAM_PARAMETER_CHECK return code.

Format

```
CALL XCTRRM(resource_ID,
            return_code)
```

Parameters

resource_ID *(input)*

Specifies the name of a resource, managed by this resource manager application, for which service is being terminated. This is a name that was specified by this application on a previous call to the Identify_Resource_Manager (XCIDRM) routine.

return_code *(output)*

Is a variable used to hold the return code passed back from the communications routine to the calling program. The *return_code* variable can return one of the following values:

- CM_OK
  Successful completion.
- CM_PROGRAM_PARAMETER_CHECK
  This virtual machine does not control the specified resource.
- CM_PROGRAM_STATE_CHECK
  This is a *TP-model application*, so this routine cannot be called.
- CM_OPERATION_NOT_ACCEPTED
  This value indicates that the *resource_ID* cannot be terminated at this time because an operation on a conversation associated with this *resource_ID* is incomplete.

State Changes

When the return code is CM_OK, **Reset** state is entered on all conversations associated with the specified *resource_ID*.

Usage Notes

1. Any resource manager application that calls the Identify_Resource_Manager routine should be sure to call Terminate_Resource_Manager before exiting. Failure to call Terminate_Resource_Manager will result in the name of the resource remaining active until CMS end-of-command processing.
2. Generally, an application should deallocate all conversations associated with the specified resource ID before calling Terminate_Resource_Manager.
3. Upon successful completion of this call, the specified resource ID is no longer identified.

Wait_on_Event (XCWOE)

An application uses the Wait_on_Event (XCWOE) routine to wait for communications from one or more partners. This routine is basically a way to handle “interrupts” from partner programs and system functions; a program can issue Wait_on_Event, then take action according to the type of interrupt it receives.

Format

```
CALL XCWOE(resource_ID,
           conversation_ID,
           event_type,
           event_info_length,
           event_buffer,
           return_code)
```

Parameters

resource_ID *(output)*

Is a variable used to return one of the following, depending on the value returned in the *event_type* parameter:

- If the *event_type* is XC_ALLOCATION_REQUEST or XC_RESOURCE_REVOKED, the *resource_ID* returns the name of a resource managed by this resource manager application for which an event has completed. The value returned is a name that was specified by this application for the *resource_ID* parameter on a previous call to Identify_Resource_Manager (XCIDRM).
- If the *event_type* is XC_USER_EVENT, the *resource_ID* returns an event identifier, specified as the *event_ID* parameter on the Signal_User_Event (XCSUE) call corresponding to this event.
- If the *event_type* is XC_INFORMATION_INPUT, XC_CONSOLE_INPUT, or XC_REQUEST_ID, the *resource_ID* parameter should not be examined.

conversation_ID *(output)*

Is a variable used to identify the conversation on which data or status information is available to be received.

This parameter is valid only when the *event_type* is XC_INFORMATION_INPUT. The contents of this variable should not be examined if *event_type* contains any other value.

event_type *(output)*

Is a variable used to indicate the type of event. The *event_type* variable can return one of the following values:

- XC_ALLOCATION_REQUEST
  A program is attempting to allocate a conversation with the application that called Wait_on_Event. The application should call Accept_Conversation (CMACCP) to process this event.
- XC_INFORMATION_INPUT
  A partner program is attempting to communicate information to the application that called Wait_on_Event. For instance, it might be sending data or deallocating the conversation. The application should call Receive (CMRCV) to process this event.

- XC_RESOURCE_REVOKED
  Another program has revoked the resource being managed by the application that called Wait_on_Event.
- XC_CONSOLE_INPUT
  Information is available from the console attached to this virtual machine. This information is placed in the *event_buffer* parameter.
- XC_REQUEST_ID
  A Shared File System (SFS) asynchronous event has completed. The request ID is placed in the *event_info_length* parameter.
- XC_USER_EVENT
  An interrupt handler or other program in the local program's virtual machine called Signal_User_Event. The values specified for the Signal_User_Event input parameters are returned by the following Wait_on_Event output parameters:

XCSUE (input)	XCWOE (output)
event_ID	*resource_ID*
user_data	*event_buffer*
user_data_length	*event_info_length*

event_info_length *(output)*
Is a variable whose contents depend on the *event_type* parameter as follows:

- If the *event_type* is XC_INFORMATION_INPUT, this variable indicates the number of data bytes that are available to be received. See Usage Note 2 on page 575 for details of how to use this value on a Receive call.
- If the *event_type* is XC_CONSOLE_INPUT or XC_USER_EVENT, this variable indicates how many bytes of data are available in the *event_buffer* parameter.
- If the *event_type* is XC_REQUEST_ID, this variable contains the actual request ID.
- If the *event_type* is XC_ALLOCATION_REQUEST or XC_RESOURCE_REVOKED, the contents of this variable should not be examined.

event_buffer *(output)*
Is the name of a 130-byte character string buffer. If more than 130 bytes are supplied, CPI Communications uses only the first 130 bytes.

The contents of this buffer depend on the value returned in the *event_type* parameter:

- If *event_type* is XC_CONSOLE_INPUT, this buffer contains the contents of the console input buffer.
- If the *event_type* is XC_USER_EVENT, this buffer contains the user event data as specified in the *user_data* parameter on the Signal_User_Event call corresponding to this event.
- If the *event_type* is XC_ALLOCATION_REQUEST, XC_INFORMATION_INPUT, XC_RESOURCE_REVOKED, or XC_REQUEST_ID, this variable should not be examined.

The *event_info_length* parameter returns the number of bytes stored in this buffer.

return_code *(output)*

Is a variable used to hold the return code passed back from the communications routine to the calling program. The *return_code* variable can return one of the following values:

- CM_OK
  Successful completion.
- CM_PRODUCT_SPECIFIC_ERROR
  This return code can result from one of the following conditions:
  - There was a problem attempting to obtain or to release storage. A file named CPICOMM LOGDATA A is appended with one of the following lines:

    ```
    XCWOE_PRODUCT_SPECIFIC_ERROR:  Unable to get storage
    ```

    ```
    XCWOE_PRODUCT_SPECIFIC_ERROR:  Unable to free storage
    ```

  - There was a problem on a call to the callable services library (CSL) routine DMSCHECK. If the problem was encountered while calling DMSCHECK, a file named CPICOMM LOGDATA A is appended with the following line:

    `XCWOE_PRODUCT_SPECIFIC_ERROR:  Call to DMSCHECK failed with CSL return code` *dd*

    where *dd* is a negative value. If the DMSCHECK CSL routine encountered the problem, CPICOMM LOGDATA A is appended with the following line:

    `XCWOE_PRODUCT_SPECIFIC_ERROR:  Call to DMSCHECK returned reason code` *reascode*

    where *reascode* is the reason code returned.

    Possible return codes and reason codes are described in the *VM/ESA: CMS Application Development Reference*.

- CM_PROGRAM_STATE_CHECK
  This value indicates that a successful call has not been made to Accept_Conversation (CMACCP), Initialize_Conversation (CMINIT), or Identify_Resource_Manager (XCIDRM) prior to calling Wait_on_Event (XCWOE).

State Changes

This call does not cause a state change.

Usage Notes

1. Events are reported in this order of priority:

   a. User event
   b. Allocation request
   c. Information input
   d. Resource revoked notification
   e. Request ID
   f. Console input

   For example, as long as there are user events pending, they will be reported first. Within an event type, however, events may not be reported in the order in which they occurred.

- **User event**:

  The application determines how to interpret the user event and what further action to take. User events are reported first in, first out (FIFO). This information will not be available after it has been presented to the program.

- **Allocation request**:

  After receiving the allocation event indication, the application can call the Accept_Conversation routine, which will establish the conversation and assign a *conversation_ID*. If necessary, the resource manager application can get information about the conversation by calling the appropriate Extract routines after accepting the conversation.

  This event will be reported by Wait_on_Event calls until an Accept_Conversation (CMACCP) call is issued to process this event or until Terminate_Resource_Manager (XCTRRM) is called to end management of the subject resource.

- **Information input**:

  After getting this event, the application should call the Receive routine. Usage Note 2 on page 575 describes how to use the value returned in the *event_info_length* parameter for the *requested_length* on the Receive call. Note that for protected conversations (*sync_level* of CM_SYNC_POINT), if the application initiates a backout sync point instead of issuing a Receive, the data or information associated with the event is purged unless the information is deallocation notification. This event will be reported by Wait_on_Event calls until a Receive (CMRCV) call is issued 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. If there are multiple conversations with the information input event, this event will not necessarily be reported on every Wait_on_Event call.

- **Resource revoked notification**:

  The application will no longer receive allocation requests for this resource. No new connections may be made to the specified *resource_ID*. This resource ID is no longer known to the rest of the system or TSAF or CS collection. Existing conversations for this resource ID are not affected. The actions taken for this event are application-specific. The resource manager still must issue Terminate_Resource_Manager after all associated conversations have been deallocated. This information will not be available after it has been presented to the program.

- **Request ID**:

  After receiving the request ID event indication, the application should examine the *event_info_length* field to get the request ID for a Shared File System asynchronous event. This information will not be available after it has been presented to the program. For information on SFS asynchronous events, see the *VM/ESA: CMS Application Development Guide*.

- **Console input**:

  The application determines how to interpret what is input from the console and what further action to take. The console input will not be available after it has been presented to the program.

2. The *event_info_length* parameter indicates how much data is available when the *event_type* is XC_INFORMATION_INPUT, XC_CONSOLE_INPUT, or XC_USER_EVENT.

   When the *event_type* is XC_INFORMATION_INPUT, the value returned by *event_info_length* can be used for the *requested_length* parameter on a subsequent call to the Receive routine to receive the data. 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.

   For mapped conversations, the value returned by *event_info_length* may be greater than the number of bytes sent by the remote program. Using this length on the call to Receive will not guarantee that the Receive call will complete immediately; the only way to guarantee that a call to Receive will complete immediately is to set *receive_type* to CM_RECEIVE_IMMEDIATE prior to calling Receive.

3. This call causes the entire application to wait. Multitasking applications should use the CMS event management services to allow a single thread to wait for an event while the others continue processing.

VM/ESA Variables and Characteristics

The following tables are provided for the variables and characteristics used with the VM/ESA extension calls shown in this appendix:

- A chart showing the possible values for variables and characteristics associated with VM/ESA extension calls. The valid pseudonyms and corresponding integer values are provided for each variable or characteristic.
- The data definitions for types and lengths of all VM/ESA extension characteristics and variables.

Pseudonyms and Integer Values

Values for VM/ESA variables and conversation characteristics are shown as pseudonym character strings rather than integer values. For example, instead of stating that the variable *conversation_security_type* is set to an integer value of 0, this appendix shows *conversation_security_type* being set to a pseudonym value of XC_SECURITY_NONE. Table 45 provides a mapping from valid VM/ESA pseudonyms to integer values for each variable and characteristic.

Pseudonyms can also be used for integer values in program code by making use of equate or define statements. See "Pseudonym Files" on page 530 for more information on the sample pseudonym files VM/ESA provides.

For further discussion of variables, characteristics, pseudonyms, and the various naming conventions used throughout this book, see Chapter 2, "CPI Communications Terms and Concepts" on page 17.

Table 45. VM/ESA Variables/Characteristics and Their Possible Values

Variable or Characteristic Name	Pseudonym Values	Integer Values
conversation_security_type	XC_SECURITY_NONE	0
	XC_SECURITY_SAME	1
	XC_SECURITY_PROGRAM	2
event_type	XC_ALLOCATION_REQUEST	1
	XC_INFORMATION_INPUT	2
	XC_RESOURCE_REVOKED	3
	XC_CONSOLE_INPUT	4
	XC_REQUEST_ID	5
	XC_USER_EVENT	6
resource_manager_type	XC_PRIVATE	0
	XC_LOCAL	1
	XC_GLOBAL	2
	XC_SYSTEM	3
security_level_flag	XC_REJECT_SECURITY_NONE	0
	XC_ACCEPT_SECURITY_NONE	1
service_mode	XC_SINGLE	0
	XC_SEQUENTIAL	1
	XC_MULTIPLE	2

Variable Types and Lengths

Table 46 defines the type and length of variables used specifically for VM/ESA extension calls.

Table 46. VM/ESA Variable Types and Lengths

Variable	Variable Type	Character Set	Length (in bytes)
client_user_ID	Character string	00640	8
conversation_security_type	Integer	Not applicable	4
event_buffer	Character string	No restriction	130
event_ID	Character string	No restriction	8
event_info_length	Integer	Not applicable	4
event_type	Integer	Not applicable	4
local_FQ_LU_name	Character string	00640	17
local_FQ_LU_name_length	Integer	Not applicable	4
luwid	Character string	00640	26
luwid_length	Integer	Not applicable	4
remote_FQ_LU_name	Character string	00640	17
remote_FQ_LU_name_length	Integer	Not applicable	4
resource_ID	Character string	00640	8
resource_manager_type	Integer	Not applicable	4
security_level_flag	Integer	Not applicable	4
security_password	Character string	00640	0-8
security_password_length	Integer	Not applicable	4
security_user_ID [1]	Character string	00640	0-8
security_user_ID_length	Integer	Not applicable	4
service_mode	Integer	Not applicable	4
user_data	Character string	No restriction	0-130
user_data_length	Integer	Not applicable	4
workunitid	Integer	Not applicable	4

Note:

1. Because the *security_user_ID* characteristic is an output parameter on the Extract_Conversation_Security_User_ID (XCECSU) call, the variable used to contain the output character string should be defined with a length equal to the maximum specification length.

VM/ESA Special Notes

The topics in this section cover a variety of important issues relating to the VM/ESA implementation of CPI Communications.

Program-Startup Processing

In VM/ESA, when the resource being requested is a private resource, node services retrieves the name of the program to be started from a special CMS NAMES file called $SERVER$ NAMES. VM/ESA also uses this file to enable a user to control access to private resources in the virtual machine. The file contains the names of private resources and users that are allowed to connect to them. See *VM/ESA: Connectivity Planning, Administration, and Operation* for detailed information on setting up a $SERVER$ NAMES file.

In addition, the private resource program's virtual machine requires certain statements in its CP directory and PROFILE EXEC to enable it to start CMS and act as a server virtual machine. At a minimum, the CP directory needs the following entries:

- IUCV ALLOW
- IPL CMS

The IUCV ALLOW directory control statement lets programs in other virtual machines communicate with the resource program in the server virtual machine. The IPL CMS statement causes CP to start CMS in the server virtual machine. *VM/ESA: Connectivity Planning, Administration, and Operation* describes these directory control statements in more detail.

The private server virtual machine needs the following statements in its PROFILE EXEC so it can process private resource connections after it has been autologged:

- SET SERVER ON
- SET FULLSCREEN OFF or SET FULLSCREEN SUSPEND
- SET AUTOREAD OFF

The SET SERVER command enables CMS private resource processing. The SET FULLSCREEN command ensures that session services are deactivated. The SET AUTOREAD command prevents CMS from issuing a console read immediately after command execution. This means that CMS will return to `Ready;` state following end-of-command processing and thus will be able to invoke the private resource program. For more information on these CMS commands, see the *VM/ESA: CMS Command Reference*.

End-of-Command Processing

Any unterminated resources declared using Identify_Resource_Manager (XCIDRM) are terminated at CMS end of command.

Work Units

All CPI Communications conversations are associated with CMS work units in VM/ESA. This means that any events that affect CMS work units also affect associated conversations. For example, during end-of-work unit processing, all CPI Communications conversations associated with that work unit are deallocated. Work units are discussed in detail in the *VM/ESA: CMS Application Development Guide*.

External Interrupts

For CPI Communications to work properly in VM/ESA, external interrupts must be enabled for a user's virtual machine. Except for the VM/ESA extension routine Signal_User_Event (XCSUE), CPI Communications routines should not be called from an interrupt handler.

Coordination with the SAA Resource Recovery Interface

VM/ESA supports the SAA resource recovery interface for protected conversations and product-specific protected resources, such as the CMS Shared File System (SFS). See “Support for Resource Recovery Interfaces” on page 54 for an overview of CPI Communications support for the SAA resource recovery interface. The CMS Coordinated Resource Recovery (CRR) facility implements the SAA resource recovery interface in VM/ESA. CRR is available only in CMS environments. For more information on using the SAA resource recovery interface in VM/ESA, see the VM/ESA appendix to the *SAA CPI Resource Recovery Reference*. For more information on CRR, see the *VM/ESA: CMS Application Development Guide* and *VM/ESA: SFS and CRR Planning, Administration, and Operation*.

Additional Conversation Characteristics

In VM/ESA, CPI Communications maintains an additional set of characteristics for each conversation used by a program. These characteristics provide security parameters, which the program can modify.

Table 47 provides a comparison of the security conversation characteristics and initial values as set by the Initialize_Conversation call and the Accept_Conversation call.

Table 47 (Page 1 of 2). VM/ESA Security Characteristics and Their Default Values

Name of Characteristic	Initialize_Conversation sets it to:	Accept_Conversation sets it to:
conversation_security_type	The security type from side information referenced by *sym_dest_name*; if the security type is not listed in side information or if a blank *sym_dest_name* is specified on the call, *conversation_security_type* is set to XC_SECURITY_SAME.	Not applicable
security_password	The password from side information referenced by *sym_dest_name* when the *conversation_security_type* is XC_SECURITY_PROGRAM and a user ID is listed; otherwise a null string. If a blank *sym_dest_name* is specified, *security_password* is set to a null string.	Not applicable
security_password_length	The length of *security_password*, if that characteristic is assigned a value; otherwise, zero. If a blank *sym_dest_name* is specified, *security_password_length* is set to zero.	Not applicable

Table 47 (Page 2 of 2). VM/ESA Security Characteristics and Their Default Values

Name of Characteristic	Initialize_Conversation sets it to:	Accept_Conversation sets it to:
security_user_ID	The user ID from side information referenced by *sym_dest_name* when the *conversation_security_type* is XC_SECURITY_PROGRAM; otherwise a null string. If a blank *sym_dest_name* is specified, *security_user_ID* is set to a null string.	The access security user ID for the session on which the conversation startup request arrived
security_user_ID_length	The length of *security_user_ID*, if that characteristic is assigned a value; otherwise, zero. If a blank *sym_dest_name* is specified, *security_user_ID_length* is set to zero.	The length of *security_user_ID*, if that characteristic is assigned a value; otherwise zero

TP-Model Applications in VM/ESA

Recall from Chapter 2 that SAA CPI Communications provides a programming interface to IBM's SNA LU 6.2. The set of calls defined by SAA, however, does not implement every aspect of the LU 6.2 protocol. VM/ESA provides extensions to SAA CPI Communications to support several additional LU 6.2 features, such as support for security types. VM/ESA also provides routines that are 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 section describes how CPI Communications programs in the VM/ESA environment can establish conversations that closely conform to the LU 6.2 model for communications. Such programs 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 program is also allowed to call most of the VM/ESA extension routines.

LU 6.2 Communications Model

In LU 6.2, LUs initiate and run transaction programs. A transaction program (TP) initiates a conversation with its TP partner using the services of the LUs. In Figure 27, TP A in LU x allocates a conversation to TP B in LU y. LU x formats and presents a conversation startup request in the form of an LU 6.2 Attach to LU y. LU y validates the Attach and starts a new instance of TP B.

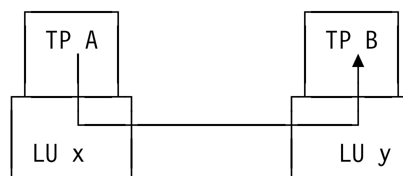


Figure 27. LU 6.2 Communications Model

In this example, both TP A and TP B are TP-model applications:

- TP A is the initial program of a distributed application. It is invoked by some process other than an Attach, typically by an end-user command. TP A has no incoming conversations.

- TP B is invoked as a result of the Attach presented to LU y. There is one and only one incoming conversation.

TP A and TP B can allocate any number of conversations with other partner programs.

VM/ESA TP-Model Applications

In Figure 28, assume that the allocation of a conversation by program A in virtual machine VMUSR1 in LU x causes program B to be automatically started by CMS in virtual machine VMUSR2 in LU y.

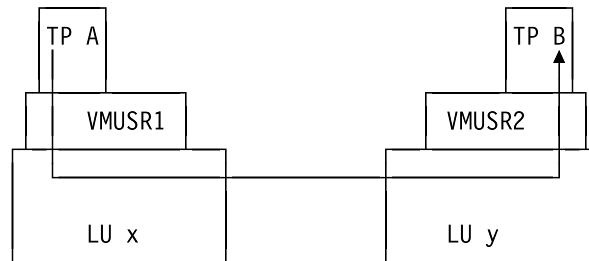


Figure 28. Creating a TP-Model Application in VM/ESA

The CPI-C calls issued by each program determine if it is a TP-model application. Program A is considered a TP-model application until it issues Identify_Resource_Manager (XCIDRM) VM/ESA extension call. Program B's classification, though, is determined by how it is started and by the first successful CPI Communications call that it issues. Calling Accept_Conversation (CMACCP) or Initialize_Conversation (CMINIT) results in program B being a TP-model application. If it is not desirable for program B to be a TP-model application, then it should issue Identify_Resource_Manager as its first CPI Communications call.

VM/ESA considers an application program like program A in Figure 28 to be a TP-model application if it has the following characteristics:

- The program is started by CMS as a result of local (end-user) action.
- There are no incoming conversations. However, the application program can allocate any number of conversations with other partner programs.
- The Identify_Resource_Manager call has not been issued successfully.

VM/ESA considers an application program like program B in Figure 28 to be a TP-model application if it has the following characteristics:

- The program is started automatically by CMS as a result of a private resource conversation startup request.
- There is only one incoming conversation - the one that caused CMS to start the program. The program can allocate any number of conversations with other partner programs, though.
- The first successful CPI Communications call is either Accept_Conversation or Initialize_Conversation.

VM/ESA considers an application program to be a non TP-model application if it has the following characteristics:

- The program has successfully called Identify_Resource_Manager at least once to identify a resource ID it wishes to manage.
- The program can accept multiple conversations and can also allocate any number of conversations with other partner programs.
- The program can call the VM/ESA extensions Terminate_Resource_Manager (XCTRRM) and Set_Client_Security_User_ID (XCSCUI).

Implications

Certain behavior is enforced and certain services are provided by VM/ESA for TP-model applications:

- For TP-model applications, there are three VM/ESA extension calls intended for non TP-model applications that the application programs are not allowed to issue. These functions are: Identify_Resource_Manager (XCIDRM), Terminate_Resource_Manager (XCTRRM), and Set_Client_Security_User_ID (XCSCUI).
- The virtual machine in which a TP-model application is running may require authorization to issue DIAGNOSE code X'D4'. This authorization is necessary if the TP-model application is an intermediate server allocating a conversation with a *conversation_security_type* characteristic of XC_SECURITY_SAME, which is the default value. In this case, the access security user ID provided by the incoming conversation that caused CMS to start the program is automatically propagated on the allocated conversation.

  See the *VM/ESA: CMS Application Development Guide* for more information on intermediate servers and the propagation of access security user IDs.
- For CPI Communications protected conversations (those with the *sync_level* characteristic set to CM_SYNC_POINT), screening is performed for TP-model applications to prevent allocation wrapback. Allocation wrapback occurs when a program tries to allocate a protected conversation whose logical unit of work identifier (LUWID) is already associated with another protected conversation to which the remote program is a partner. Screening is necessary because allocation wrapback can result in deadlock during sync-point processing.

  For example, assume that protected conversations between program A and program B and between program B and program C have been established, as illustrated by the solid lines in Figure 29 on page 583. Three cases of allocation wrapback are illustrated by the dotted lines in that figure. In all of these cases, allocations are being attempted for protected conversations with the same LUWID. Note that the work unit in effect when Allocate (CMALLC) is called determines the LUWID for a protected conversation. The *VM/ESA: CMS Application Development Guide* describes the relationship between LUWIDs and CMS work units.

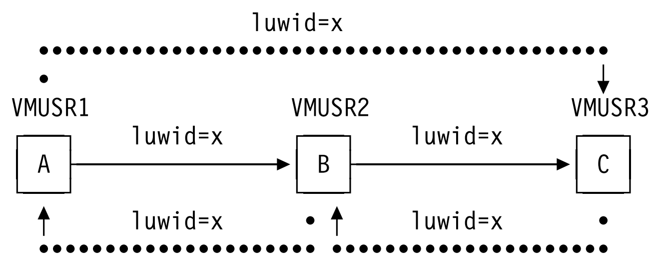


Figure 29. Three Potential Conversation Wrap-Back Scenarios

A conversation startup request that would result in allocation wrapback is automatically rejected for TP-model applications by CMS in the target virtual machine. If an application program allocates a protected conversation that would result in allocation wrapback, it receives a *return_code* value of CM_TP_NOT_AVAILABLE_NO_RETRY on a CPI Communications call following the allocate.

Certain behavior is enforced by VM/ESA for non TP-model applications:

- In order to accept any incoming conversations, the program must call the Identify_Resource_Manager function for all of the resource IDs it wishes to manage (for example, program B in Figure 28 on page 581 must call the Identify_Resource_Manager function for the resource ID of the conversation startup request from program A; otherwise, program B will not be able to accept the conversation startup request from program A).

VM/ESA-Specific Notes for CPI Communications Routines

This section contains VM/ESA-specific notes for certain CPI Communications routines.

Notes for Accept_Conversation (CMACCP):

- In VM/ESA, CPI Communications maintains additional conversation characteristics that provide security information. Table 47 on page 579 lists these additional conversation characteristics and describes how they are initialized by VM/ESA when the Accept_Conversation (CMACCP) call completes successfully.
- When accepting a conversation, the program can become a TP-model application. Refer to "TP-Model Applications in VM/ESA" on page 580 for information on how Accept_Conversation can determine whether a program is a TP-model application.

Notes for Allocate (CMALLC):

- If the remote program is within the same TSAF or CS collection, the operation of the Allocate (CMALLC) call differs from that described in Chapter 4 of this book in the following ways:
  - The Allocate (CMALLC) call does not establish an LU 6.2 session.
  - Because sessions are not used, the *return_control* conversation characteristic has no effect.
  - There is no session buffering; the Allocate (CMALLC) call can be considered to include the function of the Flush (CMFLUS) call. Therefore, the conversation startup request is flowed to the partner immediately.

  – The Allocate (CMALLC) call does not complete immediately. Instead, it completes when the remote program or VM/ESA system either accepts or deallocates the conversation startup request.

- The virtual machine in which the program is running may require authorization to issue DIAGNOSE code X'D4' to allow CPI Communications to set an alternate user ID. This authorization is usually privilege class B, unless the default privilege classes assigned to your installation have been changed.

  DIAGNOSE code X'D4' authorization is typically required when a program either is a TP-model intermediate server or issues the Set_Client_Security_User_ID (XCSCUI) call.

  If not authorized, the Allocate call will complete with a CM_PRODUCT_SPECIFIC_ERROR return code.

Note for Flush (CMFLUS):

- A Flush (CMFLUS) call has no effect when issued on a conversation that is allocated to a target program within the same TSAF or CS collection because information is not buffered for this environment. However, inclusion of Flush (CMFLUS) calls should be considered to make an application program portable.

Notes for Initialize_Conversation (CMINIT):

- If the value specified for *sym_dest_name* does not match an entry in side information, VM/ESA does not return CM_PROGRAM_PARAMETER_CHECK. Rather, the specified value is used to set the *TP_name* characteristic for the conversation. Likewise, if a matching entry is found in the side information, but no *TP_name* characteristic (:tpn. field) is associated with the entry, the specified *sym_dest_name* is used. This means that VM/ESA handles both of these cases as initializations for a conversation to a local, global, or system resource that has been identified with a name matching the specified *sym_dest_name*. Conversation characteristics are initialized to the values listed in the section entitled “Conversation Characteristics” on page 33 except as listed here.

Conversation Characteristic	Initialized Value
mode_name	set to a null string
mode_name_length	set to zero
partner_LU_name	set to a null value, which indicates *IDENT
partner_LU_name_length	set to zero
TP_name	set to the name specified in the *sym_dest_name* parameter
TP_name_length	set to the length of the name passed in the *sym_dest_name* parameter.

- In VM/ESA, CPI Communications maintains additional conversation characteristics that provide security information. Table 47 on page 579 lists these additional conversation characteristics and describes how they are initialized by VM/ESA when the Initialize_Conversation (CMINIT) call completes successfully.

- When initializing a conversation, the program can become a TP-model application. Refer to “TP-Model Applications in VM/ESA” on page 580 for

information on how Initialize_Conversation may determine whether a program is a TP-model application.

Notes for Send_Data (CMSEND):

- If Send_Data (CMSEND) is called from a REXX program, the buffer parameter specified on the call cannot contain more than 32767 bytes of data. A data field exceeding this size must be partitioned into units of 32767 bytes or less. This restriction applies only to REXX.
- Setting the *send_type* conversation characteristic to CM_SEND_AND_FLUSH or CM_BUFFER_DATA does not affect a Send_Data (CMSEND) call issued on a conversation that is allocated to a target program within the same TSAF or CS collection because information is not buffered for this environment. However, setting the *send_type* characteristic should be considered in case the application is ported to another environment.
- If the conversation partner is located on the same VM/ESA system and the conversation is not allocated through the SNA network, completion of the Send_Data call depends on the partner to receive the data, to send an error notification, or to deallocate its side of the conversation abnormally. In this situation, if the conversation partner does not respond, the Send_Data call will not complete.

  Single-threaded applications that use two conversations to simulate full-duplex behavior can experience a deadlock if both partners are waiting for the other partner to respond to a Send_Data call. To avoid this problem, these applications should use multiple threads and CMS multitasking services.

Note for Set_Log_Data (CMSLD):

- The error information supplied in the *log_data* parameter is formatted by the sending LU into an Error Log general data stream (GDS) variable. The data is placed in the message text portion of the Error Log GDS variable created by the LU. The LU formats the GDS variable, filling in the appropriate length fields and the Product Set ID portion of the GDS variable. See *VM/ESA: CP Programming Services* for the format of the Log Data GDS variable in VM/ESA.

VM/ESA Communications Events

The occurrences of 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 CMS 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 VMCPIC:

- Allocation requests
- Information input from partner
- Resource revoked notification.

The other two events supported by Signal_User_Event (XCSUE) and Wait_on_Event (XCWOE), namely 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 wait on console input events. Additionally, an application can use the event management functions to generate and process its own user events.

See *VM/ESA: CMS Application Multitasking* for a full description of event management services.

The VMCPIC Event

When VMCPIC is signalled, event data is associated with the signal, and a portion of this event data forms the event key. The following sections describe the VMCPIC event data and event keys in more detail and provide some information on creating match keys.

Event Data and Keys: The following list contains a description of the event data available when VMCPIC is signalled and the event keys that are carried by the VMCPIC signals.

- Allocation request

  The event data associated with a signal for an allocation request consists of X'00000001' concatenated with the *resource_ID*:

X'00000001'	resource_ID
4 bytes	8 bytes

  The event key is composed of both fields. However, note that since allocation requests are presented in first in, first out (FIFO) order, an application should not create a key to handle allocation requests selectively based on the *resource_ID*. An application that attempts to use selective allocation request monitors may not perform as desired. The application instead should specify the match-all wildcard '*' (X'5C') for the resource ID portion of the key.

- 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*:

X'00000002'	conversation_ID	event_info_length
4 bytes	8 bytes	4 bytes

  Note: The value returned by *event_info_length* may be greater than the amount of data sent by the remote partner. An application that uses this value to set the *requested_length* parameter on a Receive call may need to verify that it does not exceed the maximum length allowed for a single Receive call.

  The event key consists of the first two fields.

- Resource revoked notification

  The event data associated with a signal for a resource revoked notification consists of X'00000003' concatenated with the *resource_ID*:

X'00000003'	resource_ID
4 bytes	8 bytes

  The event key contains both fields.

Creating Match Keys: An application may specify either a key that exactly matches the signal key of interest, or a partial key, possibly including wildcard characters, which matches a broader range of occurrences.

The *event_type* values provided in the VM/ESA pseudonym files may be used to build keys or to determine which event was signalled by a VMCPIC event; however, some conversion of the pseudonym value or event data by the application may be required.

For example, to build the key to specify when creating a monitor for information input events associated with an established conversation represented by the identifier contained in the *conversation_ID* variable, a REXX application may use a line of code like this:

```
match_key = d2c(XC_INFORMATION_INPUT,4)||conversation_ID
```

Notes on the VMCPIC Event

Here is some information that a programmer should understand before writing an application that uses the VMCPIC event.

- Event services and the VMCPIC event are supported for the Assembler, C, and REXX programming languages.
- VMCPIC is not signalled for an allocation request event unless an Identify_Resource_Manager (XCIDRM) call was previously issued for the corresponding *resource_ID*.
- VMCPIC is not signalled for an information input event if a Receive (CMRCV) call with a *receive_type* of CM_RECEIVE_AND_WAIT is outstanding for the corresponding conversation at the time the event occurs. Thus, if both a Receive call and an EventWait for a VMCPIC monitor were waiting when an information input event occurred, only the Receive call would complete.
- VMCPIC is not signaled if Request_To_Send arrives without any accompanying data.
- The occurrence of an allocation request event or an information input event is signalled only once by VMCPIC, rather than continuously until the event has been processed as is done by the Wait_on_Event call.
- An application using the VMCPIC event should be written such that all communications-related actions are performed in response to the signalling of VMCPIC. In other words, the application using VMCPIC should not anticipate that certain communications events will happen and handle those events before VMCPIC has signalled their occurrence.

Using the Online HELP Facility

You can receive online information about the CPI Communications calls and the VM/ESA extensions described in this book using the VM/ESA HELP Facility. For example, to display a menu of callable routines, enter:

```
help routine menu
```

To display a list of general tasks for which callable routines exist, enter

```
help routine task
```

To display information about a specific call (CMINIT in this example), enter:

```
help routine cminit
```

For more information about using the HELP Facility, see the *VM/ESA: CMS User's Guide*. To display the main HELP Task Menu, enter:

```
help
```

For more information about the HELP command, see the *VM/ESA: CMS Command Reference* or enter:

```
help cms help
```

Chapter 13. CPI Communications on IBM eNetwork Personal Communications V4.1 for Windows 95

This chapter contains information about the CPI Communications implementation included in IBM eNetwork Personal Communications AS/400 and 3270 V4.1 for Windows 95 and eNetwork Personal Communications AS/400 V4.1 for Windows 95.

The topics covered are:

- Conformance Classes Supported
- Personal Communications V4.1 Publications
- Languages Supported
- Linking with the CPI-C Library
- Accepting Conversations
- Extension calls
- Deviations from the CPI-C architecture

Please see Chapter 14, “CPI Communications on Win32 and 32-bit API Client Platforms” on page 593 for a description of CPI-C Communications shipped with the following products:

- IBM eNetwork Communication Server for Windows NT 5.0, 5.01, and above
- Win95 API Client for Communication Server 5.0, 5.01, and above
- WinNT API Client for Communication Server 5.0, 5.01, and above
- OS/2 API Client for Communication Server 5.0, 5.01, and above

- Win95 API Client for Netware for SAA 2.2
- WinNT API Client for Netware for SAA 2.2
- OS/2 API Client for Netware for SAA 2.2

- Win95 API Client for IntraNetware for SAA 2.3, 3.0 and above
- WinNT API Client for IntraNetware for SAA 2.3, 3.0 and above
- OS/2 API Client for IntraNetware for SAA 2.3, 3.0 and above

- IBM eNetwork Personal Communications 4.1 for WinNT and above
- IBM eNetwork Personal Communications 4.2 for Win95 and above

Conformance Classes Supported

Personal Communications V4.1 for Windows 95 supports the following CPI Communications V2.1 conformance classes:

- LU 6.2
- Conversations
- Callback function
- Conversation-level non-blocking
- Data Conversion
- Queue-level non-blocking
- Server
- Security

  All calls in this class are supported, but the input parameter of CM_SECURITY_PROGRAM_STRONG on the

© Copyright IBM Corp. 1996, 1998

Set_Conversation_Security_Type (CMSCST) call is not supported. If specified it will be rejected with the return code CM_PARM_VALUE_NOT_SUPPORTED.

Personal Communications V4.1 for Windows 95 Publications

The following publications contain detailed product information:

Table 48. Personal Communications V4.1 for Windows 95 Publications

Title	Form Number	Part Number
Personal Communications AS/400 and 3270 for Windows 95 Up and Running	SC31-8205-00	64H0669
Personal Communications AS/400 and 3270 for Windows 95 Reference	SC31-8206-00	64H0692
Personal Communications AS/400 and 3270 for Windows 95 Emulator Programming	SC31-8207-00	64H0693
Personal Communications AS/400 and 3270 for Windows 95 APPN Programming	SC31-8208-00	64H0694
Personal Communications AS/400 and 3270 for Windows 95 NOF Programming	SC31-8199-00	64H0695

Programming Language Support

The C language is supported. Only 32 bit API support is provided.

The pseudonym file WINCPIC.H is provided to assist in the development of programs. It contains the constant declarations and data types for each supplied and returned parameter in the CPI-C calls, and call prototypes. This file can be found in the INCLUDE subdirectory of the Personal Communications directory.

Linking with the CPI-C library

The WCPIC32.LIB file can be found in the LIB subdirectory of the Personal Communications directory.

Accepting Conversations

When an ATTACH is received, the TP name in the ATTACH is matched against TP names from the DEFINE_TPs. If a match is found, the executable name from that definition is started. If a match is not found, then the name of the executable is assumed to be the same as that which was specified in the ATTACH appended with ".EXE".

If a transaction program issues an Accept call (CMACCP) and specifies a TP name that has not previously been defined, the system performs an implicit definition of the TP and assigns default values to the parameters.

The defaults used are:

Attach timeout = 0 (no timeout is applied)

Receive Allocate timeout = 0 (no timeout is applied)

Attach Manager Loaded = Yes (the TP can be loaded by the Attach Manager)

These defaults mean that if you issue a call to CMACCP as described above it will not complete until an attempt is made to attach to the named TP, or you cancel the call.

Extension Calls supported

The following CPI-C extensions, defined in the WOSA WinSNA APIs, are supported:

- WinCPICStartup
- WinCPICCleanup
- WinCPICIsBlocking
- WinCPICSetBlockingHook
- WinCPICUnhookBlockingHook

Only WinCPICStartup and WinCPICCleanup are recommended for use. The remaining calls are provided for migration of existing applications.

In addition, the Personal Communications products support the additional call:

- XCHWND (Specify_Windows_Handle)

WinCPICStartup

```
int WINAPI WinCPICStartup(WORD, LPWCPICDATA );
```

The CPIC program would issue this call when starting up. It can be used to determine version information of the API. The call returns a pointer to a structure which contains:

```
WORD     wVersion
char     szDescription[WCPICDESCRIPTION_LEN+1]
```

WinCPICCleanup

```
bool WINAPI WinCPICCleanup(void);
```

This call is used to indicate that the CPIC program is ending.

Specify_Windows_Handle (XCHWND)

This function checks that no call is currently outstanding on any conversation that has conversation level non-blocking set. It then sets the global window handle which is to be used on all subsequent calls on a conversation which has conversation level non-blocking or sets non-blocking on at some time in the future. This can be reset by calling XCHWND specifying a NULL handle.

The prototype is as follows:

```
CM_ENTRY xchwnd(HWND,                    /* Window handle */
                CM_INT32 CM_PTR);        /* return code   */
```

Deviations from the CPI-C architecture

The CMTRTS call may be issued in any state of the conversation. If the conversation is not in Send, Receive, or Send-Pending state, the value returned in control_information_received has no meaning.

Chapter 14. CPI Communications on Win32 and 32-bit API Client Platforms

This chapter provides information to write applications that contain CPI Communications calls for the following platforms and products:

- IBM eNetwork Communication Server for Windows NT 5.0, 5.01, and above
- Win95 API Client for Communication Server 5.0, 5.01, and above
- WinNT API Client for Communication Server 5.0, 5.01, and above
- OS/2 API Client for Communication Server 5.0, 5.01, and above

- Win95 API Client for Netware for SAA 2.2
- WinNT API Client for Netware for SAA 2.2
- OS/2 API Client for Netware for SAA 2.2

- Win95 API Client for IntraNetware for SAA 2.3, 3.0 and above
- WinNT API Client for IntraNetware for SAA 2.3, 3.0 and above
- OS/2 API Client for IntraNetware for SAA 2.3, 3.0 and above

- IBM eNetwork Personal Communications 4.1 for WinNT and above
- IBM eNetwork Personal Communications 4.2 for Win95 and above

For the remainder of this chapter, the term "Communications CPI-C" collectively identifies these products. If a statement is not true for all these products, the exception is noted. The phrases "or later" and "or above" are used to indicate a statement is true for a particular product and its successors.

Communications CPI-C implements functions in the manner described in the main sections of the publication, except as described in “Deviations from the CPI Communications Architecture” on page 600.

This chapter is organized as follows:

- Operating Environment
  - Conformance Classes Supported
  - Languages Supported
  - CPI-C Communications Use of Environment Variables
  - Pseudonym Files
  - Defining Side Information
  - How Dangling Conversations are Deallocated
  - Scope of the Conversation_ID
  - Diagnosing Errors
  - When Allocation Requuests Are Sent
  - Deviations from the CPI Communications Architecture
- Extension Calls – System Management
- Extension Calls – Conversation
- Extension Calls – Transaction Program Control
- Special Notes

© Copyright IBM Corp. 1996, 1998

Operating Environment

The following sections explain some special considerations that should be understood when writing applications.

Conformance Classes Supported

Refer to “Functional Conformance Class Descriptions” on page 746 for a complete description of the functional conformance classes.

Communications CPI-C support the following mandatory conformance classes:

1. Conversatons
2. LU 6.2

In addition to the mandatory conformance classes listed above, the following optional conformance classes are available in Communications CPI-C:

1. Conversation-level non-blocking
2. Queue-level non-blocking
3. Server
4. Data conversation routines
5. Full-duplex conversations
6. Expedited data
7. Security
8. Callback Function
9. Secondary Information
10. Full-Duplex
11. Expedited Data

With the following exceptions:

- Win95/WinNT/OS2 API Client for Communications Server 5.0 and 5.01 does not support Full-Duplex or Expedited Data
- Win95/WinNT/OS2 API Client for Netware for SAA 2.2 does not support Full-Duplex or Expedited Data
- Win95/WinNT/OS2 API Client for IntraNetware for SAA 2.3 does not support Full-Duplex or Expedited Data

In addition to all the conformance classes listed above, Communications CPI-C supports the CPI-C extension calls documented in this chapter.

Following is a table of CPI Communication verbs which are supported by each product.

Table 49. Client Support of CPI-C Functions

Function	Long Name	Windows NT Server and Personal Comm.[4]	Windows 95 and Windows NT Clients[5]	OS/2 Clients[6]
cmaccp	Accept_Conversation	x	x	x
cmacci	Accept_Incoming	x	x	x
cmallc	Allocate	x	x	x
cmcanc	Cancel_Conversation	x	x	x
cmcfm	Confirm	x	x	x
cmcfmd	Confirmed	x	x	x
cmcnvi	Convert_Incoming	x	x	x

Table 49. Client Support of CPI-C Functions

Function	Long Name	Windows NT Server and Personal Comm.[4]	Windows 95 and Windows NT Clients[5]	OS/2 Clients[6]
cmcnvo	Convert_Outgoing	x	x	x
cmdeal	Deallocate	x	x	x
xcmdsi	Delete_CPIC_Side_Information	x	-	-
cmectx	Extract_Conversation_Context	x	x	x
xcecst	Extract_Conversation_Security_Type	x	x	x
cmecst	Extract_Conversation_Security_Type	x	x	x
cmecs	Extract_Conversation_State	x	x	x
cmect	Extract_Conversation_Type	x	x	x
xcmesi	Extract_CPIC_Side_Information	x	x	x
cmembs	Extract_Maximum_Buffer_Size	x	x	x
cmemn	Extract_Mode_Name	x	x	x
cmepln	Extract_Partner_LU_Name	x	x	x
cmesi	Extract_Secondary_Information	x	x	x
cmesui	Extract_Security_User_ID	x	x	x
cmecsu	Extract_Security_User_ID	x	x	x
xcecsu	Extract_Security_User_ID	x	x	x
cmesrm	Extract_Send_Receive_Mode	x	-	-
cmesl	Extract_Sync_Level	x	x	x
xceti	Extract_TP_ID	x	x	x
cmetpn	Extract_TP_Name	x	x	x
cmflus	Flush	x	x	x
cminit	Initialize_Conversation	x	x	x
xcinct	Initialize_Conversation_For_TP	x	x	x
cminic	Initialize_For_Incoming	x	x	x
cmptr	Prepare_To_Receive	x	x	x
cmrcv	Receive	x	x	x
cmrcvx	Receive_Expedited	x	-	-
cmrltp	Release_Local_TP_Name	x	x	x
cmrts	Request_To_Send	x	x	x
cmsend	Send_Data	x	x	x
cmsndx	Send_Expedited	x	-	-
cmserr	Send_Error	x	x	x
cmscsp	Set_Conversation_Security_Password	x	x	x
xcscsp	Set_Conversation_Security_Password	x	x	x
cmscst	Set_Conversation_Security_Type	x	x	x
xcscst	Set_Conversation_Security_Type	x	x	x
cmscsu	Set_Conversation_Security_User_ID	x	x	x
xcscsu	Set_Conversation_Security_User_ID	x	x	x
cmsct	Set_Conversation_Type	x	x	x
xcmssi	Set_CPIC_Side_Information	x	-	-
cmsdt	Set_Deallocate_Type	x	x	x
cmsed	Set_Error_Direction	x	x	x
cmsf	Set_Fill	x	x	x
cmsld	Set_Log_Data	x	x	x
cmsmn	Set_Mode_Name	x	x	x
cmspln	Set_Partner_LU_Name	x	x	x
cmsptr	Set_Prepare_To_Receive_Type	x	x	x
cmspm	Set_Processing_Mode	x	x	x
cmsqcf	Set_Queue_Callback_Function	x	x	x
cmsqpm	Set_Queue_Processing_Mode	x	x	x
cmsrt	Set_Receive_Type	x	x	x
cmsrc	Set_Return_Control	x	x	x
cmssrm	Set_Send_Receive_Mode	x	-	-
cmsst	Set_Send_Type	x	x	x

Table 49. Client Support of CPI-C Functions

Function	Long Name	Windows NT Server and Personal Comm.[4]	Windows 95 and Windows NT Clients[5]	OS/2 Clients[6]
cmssl	Set_Sync_Level	x	x	x
cmstpn	Set_TP_Name	x	x	x
cmsltp	Specify_Local_TP_Name	x	x	x
xchwnd*	Specify_Windows_Handle	x	x	-
xcstp	Start_TP	x	x	x
cmtrts	Test_Request_To_Send_Received	x	x	x
cmwcmp	Wait_For_Completion	x	x	x
cmwait	Wait_For_Conversation	x	x	x
xcendt	End_TP	x	x	x
WinCPICCleanup*		x	x	-
WinCPICIsBlocking*		-	-	-
WinCPICSetBlockingHook*		-	-	-
WinCPICStartup*		x	x	-
WinCPICUnhookBlockingHook*		-	-	-

Note:

1. *** Indicates:** WOSA function for Microsoft Windows
2. **X Indicates:** Supported Function
3. **- Indicates:** Unsupported Function
4. Windows NT Server and Personal Comm.:

   IBM eNetwork Communication Server for Windows NT 5.0, 5.01, and above

   IBM eNetwork Personal Communications 4.1 for WinNT and above

   IBM eNetwork Personal Communications 4.2 for Win95 and above
5. Windows 95 and Windows NT Clients:

   Win95 API Client for Communication Server 5.0, 5.01, and above

   WinNT API Client for Communication Server 5.0, 5.01, and above

   Win95 API Client for Netware for SAA 2.2

   WinNT API Client for Netware for SAA 2.2

   Win95 API Client for IntraNetware for SAA 2.3, 3.0 and above

   WinNT API Client for IntraNetware for SAA 2.3, 3.0 and above
6. OS/2 Clients:

   OS/2 API Client for Communication Server 5.0, 5.01, and above

   OS/2 API Client for Netware for SAA 2.2

   OS/2 API Client for IntraNetware for SAA 2.3, 3.0 and above

Languages Supported

"C" is the only language supported by Communications CPI-C.

The supplied header files and libraries needed to compile and link Communications CPI-C programs are given in the table below:

Table 49. Header Files and Libraries for CPI-C

Operating System	Header File	Library	DLL Name
WINNT & WIN95	WINCPIC.H	WCPIC32.LIB	WCPIC32.DLL
OS/2	CPIC_C.H	CPIC16.LIB or CPIC32.LIB	CPIC.DLL

CPI-C Communications Use of Environment Variables

CPI-C Communications make use of certain environment variables when processing CPI Communications calls. The environment variables become part of the process created when the CPI Communications program is started. These environment variables are:

APPCTPN
: During an Accept_Conversation (CMACCP) call, CPI-C Communications obtain the TP name from this environment variable. The CPI-C Communications completes the call when an inbound allocation request carrying the same TP name arrives, or if one is already waiting. The operator or program must set the TP name in this environment variable for an operator-started program that uses the CMACCP call. CPI-C Communications set the TP name from an inbound allocation request in this environment variable when it starts an attach manager-started program.

  Refer to “Accept_Incoming (CMACCI)” on page 600 for use of the APPCTPN environment variable with the CMACCI call.

  During an Initialize_Conversation call, CPI-C Communications obtain the local TP name from this environment variable when the call starts the TP instance. If the operator or program does not set this environment variable, CPI-C Communications use CPIC_DEFAULT_TPNAME as a default name for the local TP instance.

APPCLLU
: During an Initialize_Conversation call that starts the TP instance, CPI-C Communications obtain the local LU alias from this environment variable. The CPI-C Communications then start the TP instance on this local LU. If this environment variable is not set when the program issues an Initialize_Conversation call that starts the TP instance, CPI-C Communications start the TP instance on the default local LU configured for the node. This variable can be a maximum of eight characters.

Pseudonym Files

Integer characteristics, variables, and fields are shown throughout this chapter as having pseudonym values rather than integer values. For example, instead of stating that the variable "conversation_security_type" is set to an integer value of 0, this chapter shows "conversation_security_type" set to the pseudonym value of CM_SECURITY_NONE.

For the header file name that contains the pseudonym definitions see “Languages Supported” on page 596

Defining Side Information

The set of parameters associated with a given symbolic destination name is called a side information entry. This section provides an overview of how a Communications CPI-C user or program can add, replace, delete, and extract side information entries.

User-Defined Side Information

The Communications CPI-C side information records can be updated using the product specific configuration utility. Please consult each product's configuration documentation for this information.

Program-Defined Side Information

The programmer can write a system management program that issues Communications CPI-C calls to update the run-time side information entries and obtain the parameter values of the entries. These updates affect only the run-time information and remain in effect until changed or until the Communicaitons CPI-C subsystem is unloaded. They do not alter the configuration file.

Communications CPI-C provides calls to update the internal side information and one to extrace it. These are:

Set_CPIC_Side_Information (XCMSSI)

Add or replace the entry (all parameter values) for a symbolic destination name. See “Set_CPIC_Side_Information (XCMSSI)” on page 607 for a detailed description.

Note: This call is not supported on the following systems:

- OS2/Win95/WinNT API Clients for Communications Server 5.0 and later
- OS2/Win95/WinNT API Clients for Netware for SAA 2.2
- OS2/Win95/WinNT API Clients for IntraNetware 2.3 and later.

Delete_CPIC_Side_Information (XCMDSI)

Delete the entry for a symbolic destination name. See “Delete_CPIC_Side_Information (XCMDSI)” on page 602 for a detailed description.

Note: This call is not support on the following systems:

- OS2/Win95/WinNT API Clients for Communications Server 5.0 and later
- OS2/Win95/WinNT API Clients for Netware for SAA 2.2
- OS2/Win95/WinNT API Clients for IntraNetware 2.3 and later.

Extract_CPIC_Side_Information (XCMESI)

Return the entry for a symbolic destination name or for the nth entry. See “Extract_CPIC_Side_Information” on page 604 for a detailed description.

Note: The nth entry option is not supported on the following systems:

- OS2/Win95/WinNT API Clients for Communications Server 5.0 and later
- OS2/Win95/WinNT API Clients for Netware for SAA 2.2
- OS2/Win95/WinNT API Clients for IntraNetware 2.3 and later.

How Dangling Conversations Are Deallocated

Communications CPI-C deallocates dangling conversations as part of its cleanup processing. That is, it deallocates all remaining conversations for that program, using the "deallocate_type" or CM_DEALLOCATE_ABEND.

It is a good practice for all CPI Communications programs to deallocate all active conversations when they are finished with them. And, of course, programs that require their conversations to be deallocated with a "deallocate_type" other than CM_DEALLOCATE_ABEND must deallocate them before ending execution.

The scope of the "conversation_ID" is limited to one TP instance — that is, the TP instance with which it was associated when the conversation was initiailized.

Diagnosing Errors

Causes for the CM_PROGRAM_PARAMETER_CHECK Return Code

This section discusses the causes for the CM_PROGRAM_PARAMETER_CHECK return code that are specific to Communications CPI-C. These causes are in addition to those described in *Appendix B*, “Return Codes and Secondary Information.”

Communications CPI-C indicates the CM_PROGRAM_PARAMETER_CHECK return code because:

- The call passed a pointer to variable and the pointer is not valid.
- For non-blocking operations that initially go outstanding (CM_OPERATION_INCOMPLETE return code), if a return parameter becomes inacessible, then the CM_PROGRAM_PARAMETER_CHECK return code will be returned.

Causes for the CM_PROGRAM_STATE_CHECK Return Code

This section discusses the causes for the CM_PROGRAM_PARAMETER_CHECK return code on the Accept_Conversation (CMACCP) call that are specific to Communications CPI-C. These causes are in addition to those described in *Appendix B*, “Return Codes and Secondary Information.”

Communications CPI-C indicates the CM_PROGRAM_STATE_CHECK return code on the Accept_Conversation call because:

- The operator or program set a TP name in the APPCTPN environment variable that was incorrect; that is, it did not match the TP name on the inbound allocation request for the program

  Note: The next two items do not apply to the following systems:
  - OS2/Win95/WinNT API Clients for Communications Server 5.0 and later
  - OS2/Win95/WinNT API Clients for Netware for SAA 2.2
  - OS2/Win95/WinNT API Clients for IntraNetware 2.3 and later.
- An operator-started program issued the Accept_Conversation call, but the call expired before the inbound allocation request arrived. The duration that a call waits for an inbound allocation request is configured via the product specific configuration program.

- For an Accept_Conversation or Accept-Incoming call, at least one TP name being used by the call is a Communication CPI-C non-queued TP and there are no outstanding attaches matching any of the TP name(s) being waited on.

When Allocation Requests Are Sent

Because Communications CPI-C buffers data being transmitted to the remote LU, the allocation request generated by an Allocate call is not sent to the remote node until one of the following occurs:

- The local LU's send buffer becomes full as the result of (one or more) Send_Data calls.
- A call is executed that explicitly flushes the buffer (for example, a Flush call or a Receive call).

Deviations from the CPI Communications Architecture

CPI-C Communications support CPI Communications calls except as indicated in the following sections.

Accept_Incoming (CMACCI)

When an Accept_Incoming (CMACCI) call is issued to CPI-C Communications, the following rules are checked sequentially to determine if they apply. The TP name(s) is determined by the first applicable rule.

1. The TP name(s) from a successfully completed Specify_Local_TP_Name (CMSLTP) call for the conversation.
2. A TP name set in the APPCTPN environment variable.

If the above rules do not yield a TP name, then CMACCI completes with a CM_PROGRAM_STATE_CHECK return code.

Release_Local_TP_Name (CMRLTP)

CPI-C Communications allow the TP name XC_RELEASE_ALL on the CMRLTP. Use of this name will result in releasing all of the TP names for the program.

Mode Names Not Supported

CPI-C Communications do not allow a CPI Communication program to allocate a conversation that uses the CPSVCMG or SNASVCMG mode name. The Allocate (CMALLC) call is rejected with a return code of CM_PARAMETER_ERROR

CPI-C Communication Functions Not Available

This section lists the CPI Communications functions that are not available at the CPI Communications interface on CPI-C Communications.

Unsupported TP Names: CPI-C Communications do not support double-byte TP names and provide limited support for SNA service TP names.

Double-Byte TP Names: CPI-C Communications do not not support TP names consisting of characters from a double-byte character set, such as Kanji. These TP names begin with the `X'0E'` character and end with the `X'0F'` character. They have an even number of bytes (2 or more) between these delimiting characters.

If the program calls Allocate (CMALLC) with the *TP_name* characteristic set to a double-byte name, CPI-C Communications treat the name as ASCII and translate

each byte to EBCDIC. The resulting TP name is not valid, and the partner LU for the conversation rejects the allocation request.

SNA Service TP Names: CPI-C Communications do not support the specification of an SNA service TP name on the Set_TP_Name (CMSTPN) call, nor do they support the setting of the APPCTPN OS/2 environment variable to an SNA service TP name.

Extension Calls – System Management

A program issues system management calls to do the following:

- Set, delete, or extract parameter values as defined in the CPI-C Communications internal side information. The side information is used to assign initial characteristic values on the Initialize_Conversation (CMINIT) call. The program can also extract the current values of the side information.
- Define or delete TP definitions.

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

Table 29 lists the system management call names and briefly describes their functions.

Table 50. List of CPI-C Communications System Management Calls

Call	Pseudonym	Description
XCMDSI	Delete_CPIC_Side_Information	Deletes a side information entry for a specified symbolic destination name.
XCMESI	Extract_CPIC_Side_Information	Returns the parameter values of a side information entry for a specified symbolic destination name or entry number.
XCMSSI	Set_CPIC_Side_Information	Sets the parameter values of a side information entry for a specified symbolic destination name.
XCDEFTP	Define_TP	Defines a TP.
XCDELTP	Delete_TP	Deletes a TP definition.

Delete_CPIC_Side_Information (XCMDSI)

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

Note: This call is not supported on the following systems:

- OS2/Win95/WinNT API Clients for Communications Server 5.0 and later
- OS2/Win95/WinNT API Clients for Netware for SAA 2.2
- OS2/Win95/WinNT API Clients for IntraNetware 2.3 and later.

See “Defining Side Information” on page 443 for more information about configuring side information.

Format

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

Parameters

key *(input)*
The value of this parameter is ignored.

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

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

- CM_OK
  Successful completion.
- CM_PROGRAM_PARAMETER_CHECK
  This return code indicates one of the following:
  – The *sym_dest_name* variable contains a name that does not exist in CPI-C Communications internal side information.
  – The address of one of the variables is not valid.
- CM_PRODUCT_SPECIFIC_ERROR
  The APPC component of CPI-C Communications is not active.

State Changes

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

Usage Notes

1. If a *return_code* other than CM_OK is returned on this call, CPI-C Communications do not delete the side information entry.
2. This call does not affect any active conversation.

3. The side information is removed immediately, which affects all Initialize_Conversation calls for the deleted symbolic destination name made after completion of this call.
4. While CPI-C Communications remove the side information entry, any other program's call to change or extract the side information will be suspended until this call is completed; this suspension includes a program's call to Initialize_Conversation (CMINIT).

Extract_CPIC_Side_Information

A program issues the Extract_CPIC_Side_Information (XCMESI) call to obtain the parameter values of an entry in CPIC Communications internal side information. The program requests the entry by either an entry number or a symbolic destination name. It does not access side information in the configuration file.

See “Defining Side Information” on page 443 for information about configuring side information.

Format

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

Parameters

entry_number *(input)*

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

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

Note: This call is not supported on the following systems:

- OS2/Win95/WinNT API Clients for Communications Server 5.0 and later
- OS2/Win95/WinNT API Clients for Netware for SAA 2.2
- OS2/Win95/WinNT API Clients for IntraNetware 2.3 and later.

sym_dest_name *(input)*

Specifies the symbolic destination name of the entry, when parameter values for a named entry are needed. CPI-C Communications use this variable only when *entry_number* is 0. If *entry_number* is greater than 0, CPI-C Communications ignore this *sym_dest_name* variable.

side_info_entry *(output)*

Specifies a structure in which the parameter values are returned. The format of the structure is shown in Table 30. Values within character string fields are returned left-justified and padded on the right with space characters.

The following extended structure is available in CPI-C Communications to support 10-byte user_IDs and 10-byte user_passwords:

Table 51. Extended Entry Structure for the CPI-C Communications Call

Byte Offset	Field Length and Type	Parameter Pseudonym
0	8-byte character string	*sym_dest_name*
8	17-byte character string	*partner_lu_name*
25	3-byte character string	(reserved)
28	32-bit integer	*tp_name_type*
32	64-byte character string	*tp_name*
96	8-byte character string	*mode_name*
104	32-bit integer	*conversation_security_type*
108	10-byte character string	*security_user_ID*
118	22-byte character string (must be zeros)	(reserved)

Refer to Table 40 on page 632 for definitions of the character set usage and length of each character string parameter.

side_info_entry_length *(input)*
Specifies the length of the entry structure. Set this length to 124, or 140 for CPI-C Communications if using the extended side information stucture.

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

- CM_OK
  Successful completion.
- CM_PROGRAM_PARAMETER_CHECK
  This return code indicates one of the following:
  - The *entry_number* specifies a value greater than the number of entries in the side information.
  - The *entry_number* specifies a value less than 0.
  - The *sym_dest_name* specifies a name that is not in any entry in the internal side information, and *entry_number* specifies 0.
  - The *side_info_entry_length* contains a value other than 124 or 140.
  - The address of one of the variables is not valid.
- CM_PRODUCT_SPECIFIC_ERROR
  The APPC component of CPI-C Communications is not active.

State Changes

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

Usage Notes

1. If a *return_code* other than CM_OK is returned on this call, the values contained in the *side_info_entry* fields are not meaningful.
2. If no user ID exists in the side information, the *security_user_ID* field contains all space characters.
3. The *security_password* of the side information entry is not returned; the field (at byte offset 116) is reserved on this call, and its content is not meaningful.
4. This call does not affect any active conversation.
5. This call does not change the parameter values of the specified side information entry.
6. While CPI-C Communications extract the side information, any other program's call to change the side information is suspended until this call is completed.

   Refer to "Languages Supported" on page 437 for information on how to create the data structure using these languages.
7. The Extract_CPIC_Side_Information call and the Set_CPIC_Side_Information (XCMSSI) call use the same *side_info_entry* format. This format enables a program to obtain an entry, update a field, and restore the updated entry, provided the entry contains no *security_password*, or the program also updates the *security_password*.
8. The *entry_number* specifies an index into the current list of internal side information entries. If entries are deleted, the indexes for particular entries may change.

Set_CPIC_Side_Information

See "Defining Side Information" on page 443 for more information about configuring side information.

Note: This call is not supported on the following systems:

- OS2/Win95/WinNT API Clients for Communications Server 5.0 and later
- OS2/Win95/WinNT API Clients for Netware for SAA 2.2
- OS2/Win95/WinNT API Clients for IntraNetware 2.3 and later.

A program issues the Set_CPIC_Side_Information (XCMSSI) call to add or replace an entry in CPI-C Communications internal side information. The entry contains all the side information parameters for the conversation identified by the supplied symbolic destination name. If the entry does not exist in the side information, this call adds a new entry; otherwise, it replaces the existing entry in its entirety.

Side information in the configuration file remains unchanged. This call overrides the side information copied from the active configuration file when CPI-C Communications was started.

Format

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

Parameters

key *(input)*
The value of this parameter is ignored.

side_info_entry *(input)*
Specifies the structure containing the parameter values for the side information entry. The format of the structure is shown in Table 32. Values within character string fields must be left-justified and padded on the right with space characters.

Table 52. Entry Structure for the CPI-C Communications Set_CPIC_Side_Information Call

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

The following extended structure is available in CPI-C Communications to support 10-byte user_IDs and 10-byte user_passwords:

Table 53. Extended Entry Structure for the Communications Server Call

Byte Offset	Field Length and Type	Parameter Pseudonym
0	8-byte character string	*sym_dest_name*
8	17-byte character string	*partner_LU_name*
25	3-byte character string	(reserved)
28	32-bit integer	*TP_name_type*
32	64-byte character string	*TP_name*
96	8-byte character string	*mode_name*
104	32-bit integer	*conversation_security_type*
108	10-byte character string	*security_user_ID*
118	10yte character string	*security_password*
128	12-byte character string (must be zeroes)	(reserved)

Refer to Table 40 on page 632 for a definition of the character set usage and the length of each character string parameter.

side_info_entry_length *(input)*
Specifies the length of the entry structure. Set this length to 124, or 140 for Communications Server if using the extended side information stucture.

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

- CM_OK
  Successful completion.
- CM_PROGRAM_PARAMETER_CHECK
  This return code indicates one of the following:
  - The *sym_dest_name* field in the *side_info_entry* structure contains a space character in the left-most byte (at byte offset 0 of the structure).
  - The *tp_name_type* field in the *side_info_entry* structure specifies an undefined value.
  - The *conversation_security_type* field in the *side_info_entry* structure specifies an undefined value.
  - The *side_info_entry_length* contains a value other than 124 or 140.
  - The address of one of the variables is not valid.
- CM_PRODUCT_SPECIFIC_ERROR
  The APPC component of CPI-C Communications is not active.

State Changes

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

Usage Notes

1. If a *return_code* other than CM_OK is returned on this call, the side information entry is not created or changed.
2. The character string parameter values supplied in the fields of the *side_info_entry* structure are not checked for validity on this call. An invalid parameter value is detected later on the Allocate (CMALLC) call or on a subsequent call, such as Send_Data (CMSEND), depending on which parameter value is not valid. An invalid partner LU name or mode name is detected on the Allocate call and indicated to the program on that call. The partner LU detects an invalid TP name, user ID, or password when it receives the allocation request; in this case the partner LU returns an error indication to the program on a subsequent call following the Allocate.
3. This Set_CPIC_Side_Information call does not affect any active conversation.
4. The side information supplied on this call takes effect immediately and is used for all Initialize_Conversation calls for the new or changed symbolic destination name made after completion of this call.
5. While CPI-C Communications update the side information with the parameters from this call, any other program's call to change or extract the side information is suspended until this call is completed; this includes a program's call to Initialize_Conversation (CMINIT).
6. The Set_CPIC_Side_Information call and the Extract_CPIC_Side_Information (XCMESI) call use the same *side_info_entry* format. This format enables a program to obtain an entry, update a field, and restore the updated entry, provided the entry contains no *security_password*, or the program also updates the *security_password*.

Extension Calls—Conversation

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

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

Table 35 lists the CPI-C Communications conversation extension calls and briefly describes their functions. For the calls that set the conversation security characteristics (used with the Allocate (CMALLC) call), the program also can obtain the current values of these characteristics, except for *security_password*. This characteristic can be set, but it cannot be extracted; this restriction is intended to reduce the risk of unintentional or unauthorized access to passwords.

Table 54. List of CPI-C Communications Conversation Calls

Call	Pseudonym	Description
XCECST	Extract_Conversation_Security_Type	Returns the current value of the *conversation_security_type* characteristic.
XCECSU	Extract_Conversation_Security_User_ID	Returns the current value of the *security_user_ID* characteristic.
XCINCT	Initialize_Conv_For_TP	Initializes a new conversation for the specified TP.
XCSCSP	Set_Conversation_Security_Password	Sets the value of the *security_password* characteristic.
XCSCST	Set_Conversation_Security_Type	Sets the value of the *conversation_security_type* characteristic.
XCSCSU	Set_Conversation_Security_User_ID	Sets the value of the *security_user_ID* characteristic.

Extract_Conversation_Security_Type (XCECST)

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

Format

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

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

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

- XC_SECURITY_NONE
- XC_SECURITY_SAME
- XC_SECURITY_PROGRAM

See “Set_Conversation_Security_Type (XCSCST)” on page 616 for a description of these values.

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

- CM_OK
  Successful completion.
- CM_PROGRAM_PARAMETER_CHECK
  This return code indicates one of the following:
  – The *conversation_ID* specifies an unassigned conversation identifier.
  – The address of one of the variables is not valid.

State Changes

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

Usage Notes

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

Extract_Conversation_Security_User_ID (XCECSU)

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

The XCECSU extension call was in CPI-C Communications prior to the time the Extract_Security_User_ID (CMESUI) call was part of the CPI-C architecture. For program migration purposes, the XCECSU call continues to be supported by CPI-C Communications.

The XCECSU call provides the same function as the CMESUI call. However, it has the following differences in allowable parameters when used in releases prior to Communications Server:

1. *security_user_ID_length* can be a maximum of 8.
2. *conversation_security_type* can be set to one of the following parameters:
   - XC_SECURITY_NONE
   - XC_SECURITY_SAME
   - XC_SECURITY_PROGRAM

Initialize_Conv_For_TP (XCINCT)

A program uses the Initialize_Conv_For_TP (XCINCT) call to initialize values for various conversation characteristics before the conversation is allocated (with a call to Allocate).

XCINCT processing is similar to CMINIT processing described in "Initialize_Conversation (CMINIT)" on page 628. In addition, the XCINCT call allows the conversation being initialized to be associated with a specific TP instance.

Format

```
CALL XCINCT(conversation_ID,
            sym_dest_name,
            CPIC_TP_ID,
            return_code)
```

Parameters

conversation_ID *(output)*
Specifies the conversation identifier.

sym_dest_name *(input)*
Specifies the symbolic destination name.

CPIC_TP_ID *(input)*
Specifies the TP instance as identified by its CPIC TP ID.

If the CPIC_TP_ID is specified, the conversation being initialized is associated with that TP instance.

If the CPIC_TP_ID is set to zeros, the following rules apply:

- 0 active TP instances

  If there are no active TP instances for this process (if no prior CMACCP, CMINIT, or XCSTP call has completed successfully), the program creates a new TP instance and initializes a new conversation.

- 1 active TP instance

  If there is one active TP instance for this process, the program initializes a new conversation and associates it with that active TP instance.

- More than 1 active TP instance

  If there is more than one active TP instance for this process, a return code of CM_PRODUCT_SPECIFIC_ERROR is returned, and the program creates an error log entry.

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

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

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

Usage Notes

See Table 38 on page 628 and notes 1 through 4 on page 629.

Set_Conversation_Security_Password (XCSCSP)

A program issues the Set_Conversation_Security_Password (XCSCSP) call to set the access security password for a conversation.

The XCSCSP extension call was in CPI-C Communications prior to the time the Set_Conversation_Security_Password (CMSCSP) call was part of the CPI-C architecture. For program migration purposes, the XCSCSP call continues to be supported by CPI-C Communications.

The XCSCSP call provides the same function as the CMSCSP call.

Set_Conversation_Security_Type (XCSCST)

A program issues the Set_Conversation_Security_Type (XCSCST) call to set the security type for the conversation.

The XCSCST extension call was in CPI-C Communications prior to the time the Set_Conversation_Security_Type (CMSCST) call was part of the CPI-C architecture. For program migration purposes, the XCSCST call continues to be supported by CPI-C Communications.

The XCSCST call provides the same function as the CMSCST call.

Set_Conversation_Security_User_ID (XCSCSU)

A program issues the Set_Conversation_Security_User_ID (XCSCSU) call to set the access security user ID for a conversation.

The XCSCSU extension call was in Communications Manager prior to the time the Set_Conversation_Security_User_ID (CMSCSU) call was part of the CPI-C architecture. For program migration purposes, the XCSCSU call continues to be supported by CPI-C Communications.

The XCSCSU call provides the same function as the CMSCSU call.

Extension Calls—Transaction Program Control

Transaction program control calls permit a program to end or start a TP instance, or to determine the CPIC TP ID of a TP instance. See "TP Instances for Communications Manager" on page 625.

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

Table 36 lists the transaction program control call names and gives a brief description of their function.

Table 55. List of CPI-C Communications Transaction Program Control Calls

Call	Pseudonym	Description
XCENDT	End_TP	Ends the specified TP instance.
XCETI	Extract_TP_ID	Returns the CPIC TP ID for the specified *conversation_ID*.
XCSTP	Start_TP	Starts a new TP instance.

End_TP (XCENDT)

A program uses the End_TP (XCENDT) call to request that CPI Communications release any resources held by CPI Communications for an active TP instance, including resources held for all conversations associated with the TP instance. This call allows a reusable resource to be used consecutively among many TP instances instead of locking the resource indefinitely in CPI Communications.

When processing End_TP, CPI-C Communications issue the APPC TP_ENDED verb for the specified TP instance. Upon completion of the TP_ENDED verb, CPI-C Communications release the control blocks associated with that TP instance.

Format

```
CALL XCENDT (CPIC TP ID,
             type,
             return_code)
```

Parameters

CPIC TP ID *(input)*
Specifies the TP instance as identified by its CPIC TP ID.

When the CPIC TP ID is set to zeros, the following rules apply:

- 0 active TP instances

  If there are no active TP instances for this process (if no prior CMACCP, CMINIT, or XCSTP call has completed successfully), a return code of CM_PROGRAM_STATE_CHECK is returned.

- 1 active TP instance

  If there is one active TP instance for this process, it is ended.

- More than 1 active TP instance

  If there is more than one active TP instance for this process, a return code of CM_PRODUCT_SPECIFIC_ERROR is returned, and CPI-C Communications creates an error log entry.

type *(input)*
Specifies how resources held for a TP instance will be released. The *type* can be one of the following:

- XC_SOFT
  Specifies that the TP instance will wait for all active CPI Communications calls to complete.

- XC_HARD
  Specifies that all active CPI Communications calls for this TP instance are overridden and termination completes. It also ends the sessions being used by the conversations of that TP instance. Both sides of the conversation may receive conversation failure return codes. XC_HARD is not intended for the typical program, but for more complex CPI Communications applications.

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

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
- CM_PRODUCT_SPECIFIC_ERROR
- CM_PROGRAM_STATE_CHECK

  This value indicates CPIC TP ID is set to zeros, and there are no active TP instances.

State Changes

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

Usage Notes

None.

Extract_TP_ID (XCETI)

A program uses the Extract_TP_ID (XCETI) call to obtain the CPIC TP ID for a specified conversation.

Format

```
CALL XCETI (conversation_ID,
            CPIC TP ID,
            return_code)
```

Parameters

conversation_ID *(input)*
Specifies the conversation identifier.

CPIC TP ID *(output)*
Specifies the CPIC TP ID.

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

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

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

Usage Notes

None.

Start_TP (XCSTP)

A program uses the Start_TP (XCSTP) call to indicate to CPI Communications that a new TP instance is to be started.

Format

```
CALL XCSTP (local_LU_alias,
            local_LU_alias_length,
            TP_name,
            TP_name_length,
            CPIC TP ID,
            return_code)
```

Parameters

local_LU_alias *(input)*
CPI Communications chooses the local LU alias name by the first condition that is encountered:

1. If *local_LU_alias_length* is not 0, the *local_LU_alias* value is used.
2. If the APPCLLU environment variable exists, it is used.
3. The default LU configured for this node is used.

local_lu_alias_length *(input)*
Specifies the length of the local LU alias name.

TP_name *(input)*
CPI Communications chooses the local TP name by the first condition that is encountered:

1. If *TP_name_length* is not 0, the *TP_name* value is used.
2. If the APPCTPN environment variable exists, it is used.
3. CPIC_DEFAULT_TPNAME is used.

TP_name_length *(input)*
Specifies the length of the local TP name.

CPIC TP ID *(output)*
Specifies the CPIC TP ID.

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

- CM_OK
- CM_PROGRAM_PARAMETER_CHECK
- CM_PRODUCT_SPECIFIC_ERROR

State Changes

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

Usage Notes

The Start_TP (XCSTP) call is used to start a new TP instance. The program must use the End_TP (XCENDT) call to request that CPI Communications release resources held for that active TP instance, if it is desireable to reduce resource usage before the program ends. The Deallocate (CMDEAL) call releases resources for the specified conversation, but not for the TP instance.

Special Notes

The following sections contain information that programmers should consider when writing programs that issue CPI Communications calls.

Migration to Communications Server

When migrating from prior releases of Communications Manager to Communications Server some situations that resulted in a CM_PRODUCT_SPECIFIC_ERROR(TP_busy) return code will now complete with the return code CM_OPERATION_NOT_ACCEPTED. Refer to “TP Instances for Communications Manager” on page 625.

Multi-threaded CPI-C Programs

This section contains information that programmers should consider when writing multi-threaded programs that issue CPI-C calls to CPI-C Communications as follows:

- Scope of CPI-C conversation_ID and CPIC TP ID
  All CPI-C calls issued within one process are considered by CPI-C Communications to be part of ***one*** program. The scope of CPI-C information (such as conversation_ID or CPIC TP ID) kept for a program is the process in which it is executing. For example, if a program issues an Accept_Conversation (CMACCP) call on thread 1, the conversation_ID that is obtained can be used by that same program on thread 2 (of the same process) to issue the Send_Data call.

- Specifying TP name
  CMSLTP and CMRLTP operate on only one list of TP names that is unique per program, regardless of which thread (in that process) issues the CMSLTP or CMRLTP call. Therefore, multi-threaded CPI-C programs should coordinate the use of these calls.

  Note: When the Accept_Incoming (CMACCI) call is being used, and only one TP name needs to be specified, an alternative method is to use the Set_TP_Name (CMSTPN). Refer to “Accept_Incoming (CMACCI)” on page 600. This takes effect per conversation, and removes the need to coordinate among other threads.

- Environment variables
  When a multi-threaded operator-started program is using environment variable(s), such as `APPCTPN` or `APPCLLU` (for example for the CMACCP, CMACCI, CMINIT, XCINCT calls), setting of this variable by the program should be serialized across threads (if using a different value of the variable per thread).

- CMWCMP conficting OOIDs
  Refer to “Wait_For_Completion (CMWCMP)” on page 631.

Considerations for CPI Communications Calls

This section describes TP instances, followed by CPI Communications calls that require special consideration when one is writing a CPI Communications program to be run on a CPI-C Communications system. Each call needing special attention is discussed in alphabetical order by call name.

TP Instances for CPI-C Communications

Within a process, CPI Communications creates an executable instance of a transaction program (TP instance) when a program issues any of the following CPI Communications calls:

- Accept_Conversation (CMACCP) call

  Note: Multiple CMACCP calls can be issued within an process. Each CMACCP creates a new TP instance.

- Accept_Incoming (CMACCI) call

  Note: Multiple CMACCI calls can be issued within an process. Each CMACCI creates a new TP instance.

- Initialize_Conversation (CMINIT) call, if all of the following conditions are met (within this process):
  - No prior CMINIT call has been issued
  - No prior CMACCP calls have been issued
  - No prior XCSTP calls have been issued

- Start_TP (XCSTP) call

  Note: Multiple XCSTP calls can be issued within an process. Each XCSTP creates a new TP instance.

CPI Communications represents the TP instance by using a transaction program identifier, or CPIC TP ID. CPI Communications converts each CPI Communications call, other than Extract and Set calls, to an APPC verb and makes a call across its APPC interface to process the verb. Each CPIC TP ID is uniquely associated with an APPC TP identifier (TP ID). Each APPC verb includes the TP ID as a parameter.

CPI Communications associates access security information (if any) that it obtains when it starts the TP instance. It maintains the correlation of this information to the CPIC TP ID until the TP instance ends.

Each TP instance remains active until the program ends, or until it is explicitly ended by using the End_TP (XCENDT) call.

Each conversation is associated with only one TP instance. However, a TP instance can be associated with more than one conversation. When the TP instance ends, all associated conversations are ended.

Usage Note 1: The Initialize_Conversation (CMINIT) call is used when a program initializes a conversation within an process that contains up to one TP instance. When initializing a conversation within an process that contains more than one TP instance, the program must specify a particular TP instance (CPIC TP ID) and the conversation will be associated with it. This association is done by using the Initialize_Conv_For_TP (XCINCT) call.

Accept_Conversation (CMACCP) or Accept_Incoming (CMACCI)

When the Accept_Conversation or Accept_Incoming call completes successfully, the following conversation characteristics are initialized:

Table 56. Additional CPI-C Communications Characteristics Initialized following CMACCP or CMACCI

Conversation Characteristic	Initialized Value
conversation_security_type	CM_SECURITY_SAME (or eqivalently XC_SECURITY_SAME) **Note:** This value is set regardless of the level of access security information (if any) on the inbound allocation request.
security_user_ID	The value received on the conversation startup request. If the conversation startup request contained no access security information, this characteristic is set to a single-space character.
security_user_ID_length	The length of *security_user_ID*.
security_password	A single-space character.
security_password_length	Set to 1.

For the Accept_Conversation (CMACCP) call, CPI-C Communications process the following rules, sequentially, the first rule that is true is used to determine the TP name(s) to use for accepting an incoming conversation:

1. The TP name(s) from successfully completed Specify_Local_TP_Name (CMSLTP) call(s) for this progam. Refer to "Specify_Local_TP_Name (CMSLTP)" on page 451 for allowable TP names on the CMSLTP call.
2. A TP name set in the APPCTPN environment variable. The environment variable must be set to a valid TP name.

   Partially specified TP names and '*' (allowed on the CMSLTP call) are not allowable TP names for the APPCTPN environment variable.

If none of the above rules yields a TP name, then CMACCP completes with a CM_PROGRAM_STATE_CHECK return code.

For the rules determining the TP name used on the CMACCI call, refer to "Accept_Incoming (CMACCI)" on page 600

When an inbound allocation request arrives with this TP name specified, CPI-C Communications complete the call.

Notes:

1. When the APPCTPN environment variable is used to specify a TP name, the TP name set in the environment variable is case sensitive; that is, lowercase letters are not converted to uppercase. The TP name set in the environment variable is made up of ASCII characters; therefore, it must be an application TP name, not an SNA service TP name.
2. For an attach manager-started program, CPI-C Communications sets the APPCTPN environment variable with the TP name from the inbound allocation request for the conversation when it starts the program. It then uses the TP name from the environment variable to complete the subsequent Accept_Conversation or Accept_Incoming call from the program. Therefore, for CPI-C Communications to match the Accept_Conversation or Accept_Incoming

call with the inbound conversation, the attach manager-started program should not set the environment variable to a different TP name.

3. CPI-C Communications recognize certain error conditions while accepting conversations that are specific to their use of OS/2 environment variables. See "Diagnosing Errors" on page 448 for more details.
4. Each Accept_Conversation (CMACCP) or Accept_Incoming (CMACCI) call starts a new TP instance. The program must use the End_TP (XCENDT) call to request that CPI Communications release resources held for that active TP instance, if it is desireable to reduce resource usage before the program ends. The Deallocate (CMDEAL) call releases resources for the specified conversation, but not for the TP instance.

Extract_Conversation_Context (CMECTX)

The Extract_Conversation_Context (CMECTX) call currently returns a context that is the left justified 12-byte CPIC TP ID padded by zeros on the right, to 32-bytes total length.

Since contexts are subject to future architectural changes, the content of this field is subject to change. In addition, dependencies on the context are not portable across platforms.

Extract_Secondary_Information (CMESI)

After a call fails that causes CPI-C to generate secondary information, the next CPI-C call for that conversation should be the Extract_Secondary_Information (CMESI) call (because other calls can reset the secondary information).

If a CMACCP or CMACCI call is not successful, CPI-C will return a temporary conversation ID for use on the CMESI call. It is recommended that the CPI-C program issue the CMESI as soon as possible, since the CPI-C implementation will eventually delete the secondary information and conversation ID.

CPI-C Communications provide limited support for the CMESI call as follows:

- The only supported values of the call_ID parameter for a CMESI call are the following:
  - CM_CMACCI
  - CM_CMACCP
  - CM_CMALLC
  - CM_CMCFM
  - CM_CMDEAL
  - CM_CMPTR
  - CM_CMRCV
  - CM_CMSEND
  - CM_CMSERR

  For all other values of this parameter, the call is returned with the return code CM_NO_SECONDARY_INFORMATION.
- CMESI returns additional secondary information if the previous CMACCP, CMACCI, or CMALLC call was rejected with a CM_PRODUCT_SPECIFIC_ERROR return code and error information is available to CPI-C Communications (for example, a failure in a call to the APPC component). For errors detected by the operating system rather than CPI-C

Communications, this call returns CM_NO_SECONDARY_INFORMATION if the CPI-C Communications program has no additional information about the failure.

The layout of the “Additional information from the implementation,” which is a subfield of secondary information, is described in the CPI-C Communications C Language header file. The format of the field is determined by what type of error has occurred.

Initialize_Conversation (CMINIT)

When Initialize_Conversation completes successfully, the CPI-C Communications specific characteristics are initialized as shown in Table 38.

Table 57. Additional CPI-C Communications Characteristics Initialized following CMINIT

Conversation Characteristic	Initialized Value
conversation_security_type	Security type from side information referenced by *sym_dest_name*. If a blank *sym_dest_name* is specified, *conversation_security_type* is set to CM_SECURITY_SAME (or equivalently XC_SECURITY_SAME).
security_user_ID	User ID from side information referenced by *sym_dest_name*, if *conversation_security_type* is CM_SECURITY_PROGRAM (or equivalently XC_SECURITY_PROGRAM); otherwise, a single space character. If a blank *sym_dest_name* is specified, *security_user_ID* is set to a single space character.
security_user_ID_length	Length of security user ID, if *conversation_security_type* is CM_SECURITY_PROGRAM (or equivalently XC_SECURITY_PROGRAM); otherwise, 1. If a blank *sym_dest_name* is specified, *security_user_ID_length* is set to 1.
security_password	Password from side information referenced by *sym_dest_name* if *conversation_security_type* is CM_SECURITY_PROGRAM (or equivalently XC_SECURITY_PROGRAM); otherwise, a single space character. If a blank *sym_dest_name* is specified, *security_password* is set to a single space character.
security_password_length	Length of security password, if *conversation_security_type* is CM_SECURITY_PROGRAM (or equivalently XC_SECURITY_PROGRAM); otherwise, 1. If a blank *sym_dest_name* is specified, *security_password_length* is set to 1.

Initialize_Conversation Notes:

1. If an operator-started CPI Communications program is to be run on a local LU other than the default LU configured for the node, either the operator or the program must set the local LU name in an environment variable named `APPCLLU` before the program issues the CMINIT call.
2. The `APPCLLU` environment variable is used only for an operator-started CPI Communications program that issues the Initialize_Conversation call to start the TP instance. The LU name set in the environment variable is case sensitive; that is, lowercase letters are not converted to uppercase.
3. The `APPCTPN` environment variable is used to obtain the local TP name for any type CPI Communications program (operator-started or attach manager-started) that issues the Initialize_Conversation call to start the TP instance. The CPI-C Communications program sets the environment variable to a default local TP name of `CPIC_DEFAULT_TPNAME`. The operator or program may set the `APPCTPN` environment variable to a different TP name before the program issues the Initialize_Conversation call, if a different local TP name is desired. CPI-C Communications do not send this local TP name outside the node.
4. CPI-C Communications recognize certain error conditions on the Initialize_Conversation call that are specific to its use of environment variables. See "Diagnosing Errors" on page 599 for more details.
5. For CMINIT, the following rules apply:
   - 0 active TP instances

     If there are no active TP instances for this process (if no prior CMACCP, CMINIT, or XCSTP has completed successfully), CPI-C Communications create a new TP instance and initializes a new conversation.
   - 1 active TP instance

     If there is one active TP instance for this process, the CPI-C Communications program initializes a new conversation and associates it with that active TP instance.
   - More than 1 active TP instance

     If there is more than one active TP instance for this process, a return code of CM_PRODUCT_SPECIFIC_ERROR.

     When a CMINIT call starts a TP instance, the TP instance remains active until the End_TP (XCENDT) is issued or the program ends.

     For multiple TP instances, use an XCINCT call to initialize a new conversation.

Receive (CMRCV)

When the Receive call is receiving data from a basic conversation, the 2-byte logical record length, or LL, field of the data is in System/370 format, with the left byte being the most significant. Depending on the programming language used, the program might have to reverse the bytes to use the field value in an integer operation.

CPI-C Communications do not perform any EBCDIC-to-ASCII translation on the data before placing it in the *buffer* variable.

See “Performance Considerations For Using Send/Receive Buffers” on page 500 for additional information concerning data buffers.

Send_Data (CMSEND)

When the Send_Data call sends data on a basic conversation, the 2-byte logical record length, or LL, field of the data must be in System/370 format, with the left byte being the most significant. If the program obtains this value from an integer operation, it might have to reverse the bytes before issuing the call, depending on the programming language used.

CPI-C Communications does not perform any ASCII-to-EBCDIC translation on the data when it sends the data from the *buffer* variable.

See “Performance Considerations For Using Send/Receive Buffers” on page 500 for additional information concerning data buffers.

Send_Expedited_Data (CMSNDX)

A CMSNDX call will not complete if the partner program has not yet received the data from a prior CMSNDX call for that same conversation. This is illustrated with the following example where sequential non-blocking Send_Expedited_Data (CMSNDX) calls are issued to CPI-C Communications:

1. A program issues a CMSNDX call, with a resulting CM_OK return code.
2. A second CMSNDX call is issued immediately for that same conversation.

   If the partner program has not yet issued the Receive_Expedited_Data (CMRCVX) call to obtain the data from the first CMSNDX call, then the result of the second CMSNDX call is a CM_OPERATION_INCOMPLETE return code. The second CMSNDX call will not complete until the partner has issued the Receive_Expedited_Data call to receive the data for the first CMSNDX call.

Set_Partner_LU_Name (CMSPLN)

The program can set the *partner_LU_name* characteristic to either an alias or a network name. Alias and network names are distinguished from each other on this call as follows:

- An alias name is 1–8 characters and does not contain a period.
- A network name is 2–17 characters, with a period separating the network ID (0–8 characters) from the network LU name (1–8 characters). If the network name does not include a network ID, the period must still be inserted preceding the network LU name, to distinguish the name as a network name instead of an alias name.

Set_Queue_Processing_Mode (CMSQPM)

For the Set_Queue_Processing_Mode (CMSQPM) call, a valid memory area must be allocated for the user_field parameter, even if this parameter is not being used by the program.

Test_Request_To_Send (CMTRTS)

The CMTRTS call does not support CM_EXPEDITED_DATA_AVAILABLE.

Wait_For_Completion (CMWCMP)

For the Wait_For_Completion (CMWCMP) call a valid memory area must be allocated for the user_field_list parameter, even if this parameter is not being used by the program.

For CMWCMP, 512 is the maximum OOID_list_count that can be specified.

If on a CMWCMP call, a valid OOID (for example, OOIDx) is specified but there is no outstanding operation for that OOID (at the time the CMWCMP call is issued by the program), then the following will occur:

- If there is at least one OOID on the list for which there is an outstanding operation, then OOIDx will be ignored (treated as a NULL (integer zero) OOID).
- If there are valid OOIDs on the list, but all of them have no outstanding operations, then the following is returned: CM_PROGRAM_STATE_CHECK return code.

Conflicting OOIDs:

Suppose OOIDx has outstanding operations, and it is being waited on by a CMWCMP call (denoted CMWCMP1). Before CMWCMP1 completes, if another call (CMWCMP2) waits on OOIDx, then OOIDx will be ignored (treated as a NULL (integer zero) OOID) with respect to CMWCMP2. Moreover, OOIDx will not be waited on by CMWCMP2, even if CMWCMP1 completed for a reason other than OOIDx. To avoid this situation, do one of the following:

- Do not use the same OOID simultaneously on more than one CMWCMP call.
- Use the timeout paramater on CMWCMP to allow your program to get control to periodically reissue the CMWCMP call.

Characteristics, Fields, and Variables

This section defines the values and data types for the additional characteristics, fields, and variables used with the CPI-C Communications calls. It also includes the CPI Communications variables for which CPI-C Communications impose certain restrictions.

The following distinctions are made regarding characteristics, fields, and variables:

Characteristic

An internal parameter of a given conversation whose value is maintained within the CPI Communications component. The value of a conversation characteristic is initialized during the Initialize_Conversation (CMINIT) or Accept_Conversation (CMACCP) call for that conversation. The value may be changed subsequently using a Set call.

Field

An element of a data structure. The data structure itself is specified as a variable on the Set_CPIC_Side_Information and Extract_CPIC_Side_Information calls. A field can supply a value as input on a call, or return a value as output from a call.

Variable
A parameter specified on a call. A variable can supply a value as input on a call, or return a value as output from a call.

Note: CPI-C Communications do not support all values of the conversation characteristics and variables described in *Appendix A*, "Variables and Characteristics."

Variable Types and Lengths

The use of character set 01134 or 00640, as defined for each field or variable, is recommended for consistency with the CPI-C definition; however, CPI-C Communications do not enforce this.

Table 58. CPI-C Communications Variable and Field Types and Lengths

Variable or Field	**Data Type**	**Character Set**	**Length**
conversation_security_type	Integer	Not applicable	32 bits
CPIC TP ID	Character string	Not applicable	12 bytes
key [1, 10]	Character string	01134	8 bytes
mode_name [1, 2]	Character string	01134	0-8 bytes
partner_LU_name [3, 5] (as an alias name)	Character string	01134	1-8 bytes
partner_LU_name [1, 4] (as a network name)	Character string	01134	2-17 bytes
security_password [5]	Character string	00640	1-8, 1-10 bytes[12]
security_password_length	Integer	Not applicable	32 bits
security_user_ID [5]	Character string	00640	1-8, 1-10 bytes[13]
security_user_ID_length	Integer	Not applicable	32 bits
side_info_entry [6]	Data structure	Field dependent	124 bytes
side_info_entry_length	Integer	Not applicable	32 bits
sym_dest_name [7, 11]	Character string	01134	8 bytes
TP_name [1, 5, 8] (as an application TP name)	Character string	01134	1-64 bytes
TP_name [1, 9] (as an SNA service TP name)	Character string	01134	1-4 bytes
TP_name_type	Integer	Not applicable	32 bits

Referenced Notes:

1. The national characters @, #, and $ are also allowed as part of the key, mode name, partner LU name, and TP name. (For a TP name only, lower case alphabetic characters are also acceptable.)
2. The null string (all space characters) is a valid mode name. The program should not set the *mode_name* to CPSVCMG or SNASVCMG. Although CPI-C Communications allow the program to specify these mode names, it rejects a program's Allocate (CMALLC) call with a *return_code* of

CM_PARAMETER_ERROR if the conversation characteristic is set to either of these mode names.

3. A period character is not allowed as part of an alias partner LU name. An alias partner LU name might contain ASCII characters in the range X'21' to X'FE'; however, use of characters drawn from character set 01134 (plus the national characters @, #, and $) is recommended.
4. The period must be present in a network partner LU name because it distinguishes the name as a network name instead of an alias name. If the partner LU name does not have a network ID, the period must be the first character in the *partner_LU_name* variable or field.
5. The space character is not allowed as part of a partner LU name, security password, security user ID, or application TP name, because it is used as the fill character in the corresponding fields of the *side_info_entry* data structure.
6. The *sym_dest_name* can be specified as all space characters only on the Initialize_Conversation (CMINIT) call. On all other calls that include this variable or field, the name must be 1–8 characters long.
7. An application TP name is composed entirely of ASCII characters. It cannot be a double-byte TP name—one that has a leading X'0E' byte and a trailing X'0F' byte—because CPI-C Communications do not support double-byte TP names. CPI-C Communications convert all characters of an application TP name from ASCII to EBCDIC when it includes the TP name on an allocation request.
8. An SNA service TP name is composed of a leading SNA service TP identifier byte and 0–3 additional ASCII characters; the identifier byte has a value in the range X'00' to X'0D' and X'0F' to X'3F'. An SNA service TP name may be specified only with the Set_CPIC_Side_Information call; it cannot be specified on the Set_TP_Name call.
9. The *sym_dest_name* variable on the Delete_CPIC_Side_Information (XCMDSI) and Extract_CPIC_Side_Information (XCMESI) calls must be at least 8 bytes long. The symbolic destination name within the variable may be 1–8 characters long on these calls. If the symbolic destination name is shorter than 8 characters, it must be left-justified in the variable and padded on the right with space characters. If the variable is longer than 8 bytes, the symbolic destination name is taken from the first (leftmost) 8 bytes and the remaining bytes are ignored.

WOSA Extension Calls Supported

The following CPI-C extensions, defined in WOSA WinSNA APIs, or WinCPIC APIs, are supported:

- WinCPICStartup
- WinCPICCleanup
- XCHWND (Specify_Windows_Handle)

WinCPICStartup

```
int WINAPI WinCPICStartup(WORD, LPWCPICDATA );
```

The CPI-C program would issue this call when starting up. It can be used to determine version information of the API. The call returns a pointer to a structure that contains:

```
WORD     wVersion
char       szDescription[WCPICDESCRIPTION_LEN+1]
```

Returns

The return value indicates whether the application was registered successfully and whether the Windows CPI-C implementation can support the specified version number. It is zero if it was registered successfully and the specified version can be supported; otherwise it is one of the following return codes:

WCPICSYSNOTREADY
:Indicates that the underlying network subsystem is not ready for network communication.

WCPICVERNOTSUPPORTED
The version of Windows CPI-C support requested is not provided by this particular Windows CPI-C implementation.

WCPICINVALID
The Windows CPI-C version specified by the application is not supported by this DLL.

WinCPICCleanup

```
bool WINAPI WinCPICCleanup(void);
```

This call is used to indicate that the CPI-C program is ending.

WINCPICCleanup()

This routine should be called by an application to deregister itself from a Windows CPI-C implementation.

Syntax

```
BOOL WinCPICCleanup(void)
```

Returns.: The return value indicates whether the deregistration was successful. It is non-zero if the application was successfully deregistered; otherwise it is zero.

Specify_Windows Handle (xchwnd)

Parameters

hwndNotify *(input)*
This parameter specifies the Windows HANDLE to be notified when outstanding operation completes.

return_code *(output)*
The following are possible values:

CM_OK
The call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
The Windows HANDLE is invalid.

CM_PRODUCT_SPECIFIC_ERROR
See *Appendix A*, "Common Return Codes"

Part 4. CPI-C 2.1 Appendixes

Appendix A. Variables and Characteristics 641
Pseudonyms and Integer Values 641
Character Sets 647
Variable Types 649
Integers 649
Character Strings 649
Distinguished Name 655
Program Function Identifier (PFID) 656
PFID Assignment Algorithms 656
Program Binding 658

Appendix B. Return Codes and Secondary Information 661
Return Codes 661
Secondary Information 679
Application-Oriented Information 680
CPI Communications-Defined Information 681
CRM-Specific Secondary Information 692
Implementation-Related Information 693

Appendix C. State Tables 695
How to Use the State Tables 695
Example 696
Explanation of Half-Duplex State Table Abbreviations 697
Conversation Characteristics ( ) 697
Conversation Queues () 699
Return Code Values [ ] 700
data_received and status_received { , } 702
Table Symbols for the Half-Duplex State Table 703
Effects of Calls to the SAA Resource Recovery Interface on Half-Duplex Conversations 710
Effects of Calls on Half-Duplex Conversations to the X/Open TX Interface . 711
Explanation of Full-Duplex State Table Abbreviations 712
Conversation Characteristics ( ) 712
Conversation Queues () 713
Return Code Values [ ] 713
data_received and status_received { , } 715
Table Symbols for the Full-Duplex State Table 716
Effects of Calls to the SAA Resource Recovery Interface on Full-Duplex Conversations 723
Effects of Calls on Full-Duplex Conversations to the X/Open TX Interface . . 724

Appendix D. CPI Communications and LU 6.2 725
Send-Pending State and the error_direction Characteristic 726
Can CPI Communications Programs Communicate with APPC Programs? 727
SNA Service Transaction Programs 727
Implementation Considerations 727
Relationship between LU 6.2 Verbs and CPI Communications Calls . . . 727

Appendix E. Application Migration from X/Open CPI-C 735

© Copyright IBM Corp. 1996, 1998

Appendix F. CPI Communications Extensions for Use with DCE Directory . . . 737
Profile Object . . . 737
Server Object . . . 737
Server Group Object . . . 738
Interaction of Directory Objects . . . 738
CPI-C Name Service Interface . . . 739
CNSI Calls . . . 740
Definition of New Objects . . . 740
Terminology . . . 740
Profile Object . . . 741
Server Object . . . 741
Server Group Object . . . 741
Program Installation Object . . . 742
Encoding Method for Complex Attribute Values . . . 742
Scenarios for Use of CNSI . . . 742
(PFID, *, *) . . . 743
(PFID, SDN, *) . . . 743
(PFID, SGDN, *) . . . 743
(PFID, SDN, resID) . . . 744
(PFID, SGDN, resID) . . . 744
(PFID, PDN, *) . . . 744
(PFID, PDN, resID) . . . 744

Appendix G. CPI Communications 2.1 Conformance Classes . . . 745
Definitions . . . 745
Conformance Requirements . . . 745
Multi-Threading Support . . . 745
CPI-C 2.1 Conformance Classes . . . 745
Functional Conformance Class Descriptions . . . 746
Conversations . . . 746
LU 6.2 . . . 747
OSI TP . . . 747
Recoverable Transactions . . . 748
Unchained Transactions . . . 748
Conversation-Level Non-Blocking . . . 749
Queue-Level Non-Blocking . . . 749
Callback Function . . . 749
Server . . . 750
Data Conversion Routines . . . 750
Security . . . 750
Distributed Security . . . 751
Full-Duplex . . . 751
Expedited Data . . . 751
Directory . . . 751
Secondary Information . . . 752
Initialization Data . . . 752
Automatic Data Conversation . . . 752
Configuration Conformance Class Description . . . 753
OSI TP Addressing Disable . . . 753
Relationship to OSI TP Functional Units and OSI TP Profiles . . . 754
Conformance Class Details . . . 755

Appendix H. Solution Developers Program - Enterprise Communications Partners in Development . . . 763
Program Highlights . . . 763
Membership . . . 763

Appendix A. Variables and Characteristics

For the variables and characteristics used throughout this book, this appendix provides the following items:

- A chart showing the values that variables and characteristics can take. The valid pseudonyms and corresponding integer values are provided for each variable and characteristic.
- The character sets used by CPI Communications.
- The data definitions for types and lengths of all CPI Communications characteristics and variables.

Pseudonyms and Integer Values

As explained in “Naming Conventions—Calls, Characteristics, Variables, and Values” on page 13, the values for variables and conversation characteristics are shown as pseudonyms rather than integer values. For example, instead of stating that the variable *return_code* is set to an integer value of 0, the book shows the *return_code* being set to a pseudonym value of CM_OK. Table 59 on page 642 provides a mapping from valid pseudonyms to integer values for each variable and characteristic.

Pseudonyms can also be used for integer values in program code by making use of equate or define statements. The diskette that came with this manual provides sample pseudonym files for several programming languages. The diskette also contains an example of how a pseudonym file is used by a COBOL program.

Note: Because the *return_code* variable is used for all CPI Communications calls, Appendix B, “Return Codes and Secondary Information” provides a more detailed description of its values, in addition to the list of values provided here.

© Copyright IBM Corp. 1996, 1998

Table 59 (Page 1 of 5). Variables/Characteristics and Their Possible Values		
Variable or Characteristic Names	**Pseudonym Values**	**Integer Values**
AE_qualifier_format	CM_DN	0
	CM_INT_DIGITS	2
allocate_confirm	CM_ALLOCATE_NO_CONFIRM	0
	CM_ALLOCATE_CONFIRM	1
AP_title_format	CM_DN	0
	CM_OID	1
begin_transaction	CM_BEGIN_IMPLICIT	0
	CM_BEGIN_EXPLICIT	1
call_id [1]	CM_CMACCI	1
	CM_CMACCP	2
	CM_CMALLC	3
	CM_CMCANC	4
	CM_CMCFM	5
	CM_CMCFMD	6
	CM_CMCNVI	7
	CM_CMCNVO	8
	CM_CMDEAL	9
	CM_CMDFDE	10
	CM_CMEACN	11
	CM_CMEAEQ	12
	CM_CMEAPT	13
	CM_CMECS	14
	CM_CMECT	15
	CM_CMECTX	16
	CM_CMEID	17
	CM_CMEMBS	18
	CM_CMEMN	19
	CM_CMEPID	20
	CM_CMEPLN	21
	CM_CMESI	22
	CM_CMESL	23
	CM_CMESRM	24
	CM_CMESUI	25
	CM_CMETC	26
	CM_CMETPN	27
	CM_CMFLUS	28
	CM_CMINCL	29
	CM_CMINIC	30
	CM_CMINIT	31
	CM_CMPREP	32
	CM_CMPTR	33
	CM_CMRCV	34
	CM_CMRCVX	35
	CM_CMRLTP	36
	CM_CMRTS	37
	CM_CMSAC	38
	CM_CMSACN	39
	CM_CMSAEQ	40
	CM_CMSAPT	41
	CM_CMSBT	42

Table 59 (Page 2 of 5). Variables/Characteristics and Their Possible Values		
Variable or Characteristic Names	**Pseudonym Values**	**Integer Values**
	CM_CMSCSP	43
	CM_CMSCST	44
	CM_CMSCSU	45
	CM_CMSCT	46
	CM_CMSCU	47
	CM_CMSDT	48
	CM_CMSED	49
	CM_CMSEND	50
	CM_CMSERR	51
	CM_CMSF	52
	CM_CMSID	53
	CM_CMSLD	54
	CM_CMSLTP	55
	CM_CMSMN	56
	CM_CMSNDX	57
	CM_CMSPDP	58
	CM_CMSPID	59
	CM_CMSPLN	60
	CM_CMSPM	61
	CM_CMSPTR	62
	CM_CMSQCF	63
	CM_CMSQPM	64
	CM_CMSRC	65
	CM_CMSRT	66
	CM_CMSSL	67
	CM_CMSSRM	68
	CM_CMSST	69
	CM_CMSTC	70
	CM_CMSTPN	71
	CM_CMTRTS	72
	CM_CMWAIT	73
	CM_CMWCMP	74
	CM_CMSJT	75
	CM_CMEMID	76
	CM_CMSMID	77
	CM_CMSNDM	78
	CM_CMRCVM	79
confirmation_urgency	CM_CONFIRMATION_NOT_URGENT	0
	CM_CONFIRMATION_URGENT	1
control_information_received	CM_NO_CONTROL_INFO_RECEIVED	0
	CM_REQ_TO_SEND_RECEIVED	1
	CM_ALLOCATE_CONFIRMED	2
	CM_ALLOCATE_CONFIRMED_WITH_DATA	3
	CM_ALLOCATE_REJECTED_WITH_DATA	4
	CM_EXPEDITED_DATA_AVAILABLE	5
	CM_RTS_RCVD_AND_EXP_DATA_AVAIL	6
conversation_queue	CM_INITIALIZATION_QUEUE	0
	CM_SEND_QUEUE	1
	CM_RECEIVE_QUEUE	2
	CM_SEND_RECEIVE_QUEUE	3
	CM_EXPEDITED_SEND_QUEUE	4
	CM_EXPEDITED_RECEIVE_QUEUE	5
conversation_return_code	See *return_code*.	

Table 59 (Page 3 of 5). Variables/Characteristics and Their Possible Values

Variable or Characteristic Names	Pseudonym Values	Integer Values
conversation_security_type	CM_SECURITY_NONE	0
	CM_SECURITY_SAME	1
	CM_SECURITY_PROGRAM	2
	CM_SECURITY_DISTRIBUTED	3
	CM_SECURITY_MUTUAL	4
	CM_SECURITY_PROGRAM_STRONG	5
conversation_state	CM_INITIALIZE_STATE	2
	CM_SEND_STATE	3
	CM_RECEIVE_STATE	4
	CM_SEND_PENDING_STATE	5
	CM_CONFIRM_STATE	6
	CM_CONFIRM_SEND_STATE	7
	CM_CONFIRM_DEALLOCATE_STATE	8
	CM_DEFER_RECEIVE_STATE	9
	CM_DEFER_DEALLOCATE_STATE	10
	CM_SYNC_POINT_STATE	11
	CM_SYNC_POINT_SEND_STATE	12
	CM_SYNC_POINT_DEALLOCATE_STATE	13
	CM_INITIALIZE_INCOMING_STATE	14
	CM_SEND_ONLY_STATE	15
	CM_RECEIVE_ONLY_STATE	16
	CM_SEND_RECEIVE_STATE	17
	CM_PREPARED_STATE	18
conversation_type	CM_BASIC_CONVERSATION	0
	CM_MAPPED_CONVERSATION	1
data_received	CM_NO_DATA_RECEIVED	0
	CM_DATA_RECEIVED	1
	CM_COMPLETE_DATA_RECEIVED	2
	CM_INCOMPLETE_DATA_RECEIVED	3
deallocate_type	CM_DEALLOCATE_SYNC_LEVEL	0
	CM_DEALLOCATE_FLUSH	1
	CM_DEALLOCATE_CONFIRM	2
	CM_DEALLOCATE_ABEND	3
directory_encoding	CM_DEFAULT_ENCODING	0
	CM_UNICODE_ENCODING	1
directory_syntax	CM_DEFAULT_SYNTAX	0
	CM_DCE_SYNTAX	1
	CM_XDS_SYNTAX	2
	CM_NDS_SYNTAX	3
error_direction	CM_RECEIVE_ERROR	0
	CM_SEND_ERROR	1
expedited_receive_type	CM_RECEIVE_AND_WAIT	0
	CM_RECEIVE_IMMEDIATE	1
fill	CM_FILL_LL	0
	CM_FILL_BUFFER	1
join_transaction	CM_JOIN_IMPLICIT	0
	CM_JOIN_EXPLICIT	1
partner_ID_scope	CM_EXPLICIT	0
	CM_REFERENCE	1
partner_ID_type	CM_DISTINGUISHED_NAME	0
	CM_LOCAL_DISTINGUISHED_NAME	1
	CM_PROGRAM_FUNCTION_ID	2
	CM_OSI_TPSU_TITLE_OID	3
	CM_PROGRAM_BINDING	4

Table 59 (Page 4 of 5). Variables/Characteristics and Their Possible Values		
Variable or Characteristic Names	**Pseudonym Values**	**Integer Values**
prepare_data_permitted	CM_PREPARE_DATA_NOT_PERMITTED	0
	CM_PREPARE_DATA_PERMITTED	1
prepare_to_receive_type	CM_PREP_TO_RECEIVE_SYNC_LEVEL	0
	CM_PREP_TO_RECEIVE_FLUSH	1
	CM_PREP_TO_RECEIVE_CONFIRM	2
processing_mode	CM_BLOCKING	0
	CM_NON_BLOCKING	1
queue_processing_mode	CM_BLOCKING	0
	CM_NON_BLOCKING	1
receive_type	CM_RECEIVE_AND_WAIT	0
	CM_RECEIVE_IMMEDIATE	1
request_to_send_received [2]	CM_REQ_TO_SEND_NOT_RECEIVED	0
	CM_REQ_TO_SEND_RECEIVED	1
return_code and *conversation_return_code*	CM_OK	0
	CM_ALLOCATE_FAILURE_NO_RETRY	1
	CM_ALLOCATE_FAILURE_RETRY	2
	CM_CONVERSATION_TYPE_MISMATCH	3
	CM_PIP_NOT_SPECIFIED_CORRECTLY	5
	CM_SECURITY_NOT_VALID	6
	CM_SYNC_LVL_NOT_SUPPORTED_SYS	7
	CM_SYNC_LVL_NOT_SUPPORTED_PGM	8
	CM_TPN_NOT_RECOGNIZED	9
	CM_TP_NOT_AVAILABLE_NO_RETRY	10
	CM_TP_NOT_AVAILABLE_RETRY	11
	CM_DEALLOCATED_ABEND	17
	CM_DEALLOCATED_NORMAL	18
	CM_PARAMETER_ERROR	19
	CM_PRODUCT_SPECIFIC_ERROR	20
	CM_PROGRAM_ERROR_NO_TRUNC	21
	CM_PROGRAM_ERROR_PURGING	22
	CM_PROGRAM_ERROR_TRUNC	23
	CM_PROGRAM_PARAMETER_CHECK	24
	CM_PROGRAM_STATE_CHECK	25
	CM_RESOURCE_FAILURE_NO_RETRY	26
	CM_RESOURCE_FAILURE_RETRY	27
	CM_UNSUCCESSFUL	28
	CM_DEALLOCATED_ABEND_SVC	30
	CM_DEALLOCATED_ABEND_TIMER	31
	CM_SVC_ERROR_NO_TRUNC	32
	CM_SVC_ERROR_PURGING	33
	CM_SVC_ERROR_TRUNC	34
	CM_OPERATION_INCOMPLETE	35
	CM_SYSTEM_EVENT	36
	CM_OPERATION_NOT_ACCEPTED	37
	CM_CONVERSATION_ENDING	38
	CM_SEND_RCV_MODE_NOT_SUPPORTED	39
	CM_BUFFER_TOO_SMALL	40
	CM_EXP_DATA_NOT_SUPPORTED	41
	CM_DEALLOC_CONFIRM_REJECT	42
	CM_ALLOCATION_ERROR	43
	CM_RETRY_LIMIT_EXCEEDED	44
	CM_NO_SECONDARY_INFORMATION	45
	CM_SECURITY_NOT_SUPPORTED	46
	CM_SECURITY_MUTUAL_FAILED	47
	CM_CALL_NOT_SUPPORTED	48

Variable or Characteristic Names	Pseudonym Values	Integer Values
	CM_PARM_VALUE_NOT_SUPPORTED	49
	CM_UNKNOWN_MAP_NAME_REQUESTED	50
	CM_UNKNOWN_MAP_NAME_RECEIVED	51
	CM_MAP_ROUTINE_ERROR	52
	CM_CONVERSATION_CANCELLED	53
	CM_TAKE_BACKOUT	100
	CM_DEALLOCATED_ABEND_BO	130
	CM_DEALLOCATED_ABEND_SVC_BO	131
	CM_DEALLOCATED_ABEND_TIMER_BO	132
	CM_RESOURCE_FAIL_NO_RETRY_BO	133
	CM_RESOURCE_FAILURE_RETRY_BO	134
	CM_DEALLOCATED_NORMAL_BO	135
	CM_CONV_DEALLOC_AFTER_SYNCPT	136
	CM_INCLUDE_PARTNER_REJECT_BO	137
return_control	CM_WHEN_SESSION_ALLOCATED	0
	CM_IMMEDIATE	1
	CM_WHEN_CONWINNER_ALLOCATED	2
	CM_WHEN_SESSION_FREE	3
send_receive_mode	CM_HALF_DUPLEX	0
	CM_FULL_DUPLEX	1
send_type	CM_BUFFER_DATA	0
	CM_SEND_AND_FLUSH	1
	CM_SEND_AND_CONFIRM	2
	CM_SEND_AND_PREP_TO_RECEIVE	3
	CM_SEND_AND_DEALLOCATE	4
status_received	CM_NO_STATUS_RECEIVED	0
	CM_SEND_RECEIVED	1
	CM_CONFIRM_RECEIVED	2
	CM_CONFIRM_SEND_RECEIVED	3
	CM_CONFIRM_DEALLOC_RECEIVED	4
	CM_TAKE_COMMIT	5
	CM_TAKE_COMMIT_SEND	6
	CM_TAKE_COMMIT_DEALLOCATE	7
	CM_TAKE_COMMIT_DATA_OK	8
	CM_TAKE_COMMIT_SEND_DATA_OK	9
	CM_TAKE_COMMIT_DEALLOC_DATA_OK	10
	CM_PREPARE_OK	11
	CM_JOIN_TRANSACTION	12
sync_level	CM_NONE	0
	CM_CONFIRM	1
	CM_SYNC_POINT	2
	CM_SYNC_POINT_NO_CONFIRM	3
transaction_control	CM_CHAINED_TRANSACTIONS	0
	CM_UNCHAINED_TRANSACTIONS	1

Table 59 (Page 5 of 5). Variables/Characteristics and Their Possible Values

Notes:

1. The *call_ID* values greater than 10000 are reserved for the product extension calls.
2. Early versions of CPI-C used the *request_to_send_received* variable. CPI-C 2.0 or later programs use *control_information_received*, which is an enhanced version of the *request_to_send_received* variable.

Character Sets

CPI Communications makes use of character strings composed of characters from one of the following character sets:

- Character set 01134, which is composed of the uppercase letters A through Z and numerals 0-9.
- Character set 00640, which is composed of the uppercase and lowercase letters A through Z, numerals 0-9, and 20 special characters.
- Character set T61String, which is composed of the uppercase and lowercase letters A through Z, numerals 0-9, and many additional special characters. The most commonly used special characters are provided in Table 60. See CCITT Recommendation T.61 for other defined special characters.

These character sets, along with EBCDIC hexadecimal and graphic representations, are provided in Table 60. See the *SNA Formats* manual (GA27-3136) for more information on character sets.

Table 60 (Page 1 of 3). Character Sets T61String, 01134, and 00640

EBCDIC Hex Code	Graphic	Description	Character Set		
			T61-String	01134	00640
40		Blank	X		X
4A	[	Left square bracket	X		
4B	.	Period	X		X
4C	<	Less than sign	X		X
4D	(	Left parenthesis	X		X
4E	+	Plus sign	X		X
4F	!	Exclamation mark	X		
50	&	Ampersand	X		X
5A	]	Right square bracket	X		
5C	*	Asterisk	X		X
5D	)	Right parenthesis	X		X
5E	;	Semicolon	X		X
60	-	Dash	X		X
61	/	Slash	X		X
6B	,	Comma	X		X
6C	%	Percent sign	X		X
6D	_	Underscore	X		X
6E	>	Greater than sign	X		X
6F	?	Question mark	X		X
7A	:	Colon	X		X
7C	@	Commercial a (at sign)	X		
7D	'	Single quote	X		X
7E	=	Equal sign	X		X
7F	"	Double quote	X		X
81	a	Lowercase a	X		X
82	b	Lowercase b	X		X
83	c	Lowercase c	X		X
84	d	Lowercase d	X		X
85	e	Lowercase e	X		X
86	f	Lowercase f	X		X
87	g	Lowercase g	X		X

Table 60 (Page 2 of 3). Character Sets T61String, 01134, and 00640					
EBCDIC Hex Code	Graphic	Description	Character Set		
			T61-String	01134	00640
88	h	Lowercase h	X		X
89	i	Lowercase i	X		X
91	j	Lowercase j	X		X
92	k	Lowercase k	X		X
93	l	Lowercase l	X		X
94	m	Lowercase m	X		X
95	n	Lowercase n	X		X
96	o	Lowercase o	X		X
97	p	Lowercase p	X		X
98	q	Lowercase q	X		X
99	r	Lowercase r	X		X
A2	s	Lowercase s	X		X
A3	t	Lowercase t	X		X
A4	u	Lowercase u	X		X
A5	v	Lowercase v	X		X
A6	w	Lowercase w	X		X
A7	x	Lowercase x	X		X
A8	y	Lowercase y	X		X
A9	z	Lowercase z	X		X
BB			Vertical line	X		
C1	A	Uppercase A	X	X	X
C2	B	Uppercase B	X	X	X
C3	C	Uppercase C	X	X	X
C4	D	Uppercase D	X	X	X
C5	E	Uppercase E	X	X	X
C6	F	Uppercase F	X	X	X
C7	G	Uppercase G	X	X	X
C8	H	Uppercase H	X	X	X
C9	I	Uppercase I	X	X	X
D1	J	Uppercase J	X	X	X
D2	K	Uppercase K	X	X	X
D3	L	Uppercase L	X	X	X
D4	M	Uppercase M	X	X	X
D5	N	Uppercase N	X	X	X
D6	O	Uppercase O	X	X	X
D7	P	Uppercase P	X	X	X
D8	Q	Uppercase Q	X	X	X
D9	R	Uppercase R	X	X	X
E2	S	Uppercase S	X	X	X
E3	T	Uppercase T	X	X	X
E4	U	Uppercase U	X	X	X
E5	V	Uppercase V	X	X	X
E6	W	Uppercase W	X	X	X
E7	X	Uppercase X	X	X	X
E8	Y	Uppercase Y	X	X	X
E9	Z	Uppercase Z	X	X	X
F0	0	Zero	X	X	X
F1	1	One	X	X	X
F2	2	Two	X	X	X
F3	3	Three	X	X	X
F4	4	Four	X	X	X
F5	5	Five	X	X	X
F6	6	Six	X	X	X

Table 60 (Page 3 of 3). Character Sets T61String, 01134, and 00640					
EBCDIC Hex Code	**Graphic**	**Description**	**Character Set**		
			T61-String	**01134**	**00640**
F7	7	Seven	X	X	X
F8	8	Eight	X	X	X
F9	9	Nine	X	X	X

Variable Types

CPI Communications makes use of two variable types, integer and character string. Table 61 on page 650 defines the type and length of variables used in this document. Variable types are described below.

Integers

The integers are signed, non-negative integers. Their length is provided in bits.

Character Strings

Character strings are composed of characters taken from one of the character sets discussed in “Character Sets” on page 647, or, in the case of *buffer*, are bytes with no restrictions (that is, a string composed of characters from X'00' to X'FF').

Note: The name “character string” as used in this manual should not be confused with “character string” as used in the C programming language. No further restrictions beyond those described above are intended.

The character-string length represents the number of characters a character string can contain. CPI Communications defines two lengths for some character-string variables:

- **Minimum specification length:** The minimum number of characters that a program can use to specify the character string. For some character strings, the minimum specification length is zero. A zero-length character string on a call means the character string is omitted, regardless of the length of the variable that contains the character string (see the notes for Table 61 on page 650).
- **Maximum specification length:** The maximum number of characters that a transaction program can use to specify a character string. All products can send or receive the maximum specification length for the character string.

For example, the character-string length for *log_data* is listed as 0-512 bytes, where 0 is the minimum specification length and 512 is the maximum specification length.

If the variable to which a character string is assigned is longer than the character string, the character string is left-justified within the variable and the variable is filled out to the right with space characters (also referred to as blank characters). Space characters, if present, are not part of the character string.

If the character string is formed from the concatenation of two or more individual character strings, as is discussed in note 5 on page 653 for the *partner_LU_name*, the concatenated character string as a whole is left-justified within the variable and

the variable is filled out to the right with space characters. Space characters, if present, are not part of the concatenated character string.

Table 61 (Page 1 of 3). Variable Types and Lengths			
Variable	**Variable Type**	**Character Set**	**Length**
AE_qualifier [3], [4]	Character string	T61String	0-1024 bytes
AE_qualifier_format	Integer	N/A	32 bits
AE_qualifier_length	Integer	N/A	32 bits
allocate_confirm	Integer	N/A	32 bits
AP_title [3], [4]	Character string	T61String	0-1024 bytes
AP_title_format	Integer	N/A	32 bits
AP_title_length	Integer	N/A	32 bits
application_context_name [3], [4]	Character string	00640	0-256 bytes
application_context_name_length	Integer	N/A	32 bits
begin_transaction	Integer	N/A	32 bits
buffer [1, 2]	Character string	no restriction	0-max supported by system
buffer_length [1]	Integer	N/A	32 bits
call_ID	Integer	N/A	32 bits
callback_function	Pointer [11]	N/A	system-dependent [11]
completed_op_index_list	Array of integers	N/A	n X 32 bits
completed_op_count	Integer	N/A	32 bits
confirmation_urgency	Integer	N/A	32 bits
context_ID	Character string	no restriction	1-32 bytes
context_ID_length	Integer	N/A	32 bits
control_information_received	Integer	N/A	32 bits
conversation_ID	Character string	no restriction	8 bytes
conversation_queue	Integer	N/A	32 bits
conversation_return_code	Integer	N/A	32 bits
conversation_security_type	Integer	N/A	32 bits
conversation_state	Integer	N/A	32 bits
conversation_type	Integer	N/A	32 bits
data_received	Integer	N/A	32 bits
deallocate_type	Integer	N/A	32 bits
directory_encoding	Integer	N/A	32 bits
directory_syntax	Integer	N/A	32 bits
error_direction	Integer	N/A	32 bits
expedited_receive_type	Integer	N/A	32 bits
fill	Integer	N/A	32 bits

Table 61 (Page 2 of 3). Variable Types and Lengths			
Variable	**Variable Type**	**Character Set**	**Length**
initialization_data [3, 4]	Character string	no restriction	0-10000 bytes
initialization_data_length	Integer	N/A	32 bits
join_transaction	Integer	N/A	32 bits
log_data [3]	Character string	no restriction	0-512 bytes
log_data_length	Integer	N/A	32 bits
map_name	Character string	T61String	0-64 bytes
map_name_length	Integer	N/A	32 bits
maximum_buffer_size [2]	Integer	N/A	32 bits
mode_name [3, 4, 8]	Character string	01134	0-8 bytes
mode_name_length	Integer	N/A	32 bits
OOID	Integer	N/A	32 bits
OOID_list	Array of integers	N/A	n X 32 bits
OOID_list_count	Integer	N/A	32 bits
partner_ID	Character string	no restriction	0-32767 bytes
partner_ID_length	Integer	N/A	32 bits
partner_ID_scope	Integer	N/A	32 bits
partner_ID_type	Integer	N/A	32 bits
partner_LU_name [3, 4, 5]	Character string	01134	1-17 bytes
partner_LU_name_length	Integer	N/A	32 bits
prepare_data_permitted	Integer	N/A	32 bits
prepare_to_receive_type	Integer	N/A	32 bits
processing_mode	Integer	N/A	32 bits
queue_processing_mode	Integer	N/A	32 bits
receive_type	Integer	N/A	32 bits
received_length [2]	Integer	N/A	32 bits
request_to_send_received [10]	Integer	N/A	32 bits
requested_length [2]	Integer	N/A	32 bits
return_code	Integer	N/A	32 bits
return_control	Integer	N/A	32 bits
security_password [3]	Character string	00640	0-10 bytes
security_password_length	Integer	N/A	32 bits
security_user_ID [3, 4]	Character string	00640	0-10 bytes
security_user_ID_length	Integer	N/A	32 bits
send_length [2]	Integer	N/A	32 bits
send_receive_mode	Integer	N/A	32 bits
send_type	Integer	N/A	32 bits
status_received	Integer	N/A	32 bits

Table 61 (Page 3 of 3). Variable Types and Lengths			
Variable	**Variable Type**	**Character Set**	**Length**
sym_dest_name [3, 7]	Character string	01134	8 bytes
sync_level	Integer	N/A	32 bits
timeout	Integer	N/A	32 bits
TP_name [3, 4, 6]	Character string	T61String	1-64 bytes
TP_name_length	Integer	N/A	32 bits
transaction_control	Integer	N/A	32 bits
user_field	Character string	no restriction	8 bytes
user_field_list	Array of character strings	no restriction	n X 8 bytes

Notes:

1. When a transaction program is in conversation with another transaction program executing in an unlike environment (for example, an EBCDIC-environment program in conversation with an ASCII-environment program), *buffer* may require conversion from one encoding to the other. For character data in character data set 00640, this conversion can be accomplished by Convert_Outgoing in the sending program and by Convert_Incoming in the receiving program. The maximum allowed value of the *buffer_length* parameter on the Convert_Incoming and Convert_Outgoing calls is implementation-specific.

2. The maximum buffer size for sending and receiving data may vary from system to system. The maximum buffer size is at least 32767. For more information, refer to "Extract_Maximum_Buffer_Size (CMEMBS)" on page 173.

3. Specify these fields using the native encoding of the local system. When appropriate, CPI Communications automatically converts these fields to the correct format (EBCDIC on LU 6.2 and the negotiated transfer syntax on OSI TP) when they are used as input parameters on CPI Communications calls. When CPI Communications returns these fields to the program (for instance, as output parameters on one of the Extract calls), they are returned in the native encoding of the local system. See "Automatic Conversion of Characteristics" on page 41 for more information on automatic conversion of these fields.

   Note: An LU 6.2 CRM converts log data in character set 00640 only. To enhance program portability, it is recommended that character set 00640 be used for the *log_data* characteristic.

4. Because the *mode_name*, *partner_LU_name*, *security_user_ID*, *AE_qualifier*, *AP_title*, *application_context_name*, *context_ID*, *TP_name*, and *initialization_data* characteristics are output parameters on their respective Extract calls, the variables used to contain the output character strings should be defined with a length equal to the maximum specification length.

   Note: An LU 6.2 CRM uses character set 00640 for the *partner_LU_name* and *TP_name*. An OSI TP CRM uses character set T61 String for the *AE_qualifier*, *AP_title*, *application_context_name*, and *TP_name*. Both CRM types use character set 01134 for the *mode_name*. To enhance program portability, it is recommended that character set 01134, a subset of character sets 00640 and T61 String, be used for these characteristics.

The IMS, MVS, OS/2, and OS/400 implementations of CPI Communications allow use of the $, @, and # national characters in the *mode_name* and *partner_LU_name* fields and restrict the first character in these fields to an alphabetic or national character.

OS/400 CPI Communications does not support a *mode_name* of the characters “BLANK”. “BLANK” may be specified in the side information, but it denotes that a *mode_name* of 8 space characters be used. See Chapter 11, “CPI Communications on Operating System/400” on page 513 for more information.

The OS/2 implementation allows specification of either an alias or a network name for the *partner_LU_name* variable. OS/2 distinguishes the specification of an alias name from a network name based on the absence or presence of the period character in the name. See Chapter 10, “CPI Communications on OS/2” on page 435 for more information about alias and network names used with OS/2.

For Networking Services for Windows, if an unqualified partner LU name is specified, that name will be treated as though qualified with the network ID of the local LU name.

CPI Communications applications in CICS cannot be SNA service programs and, therefore, cannot allocate on the mode names SNASVCMG or CPSVCMG. If they attempt to do this, they will get the CM_PARAMETER_ERROR return code. For Networking Services for Windows or OS/2, the program should not set the *mode_name* conversation characteristic to CPSVCMG or SNASVCMG. Although Networking Services for Windows and OS/2 allow the program to specify these mode names on the Set_Mode_Name call, they reject the subsequent Allocate (CMALLC) call with a *return_code* of CM_PARAMETER_ERROR.

5. The *partner_LU_name* can be of two varieties:

   - A character string composed solely of characters drawn from character set 01134
   - A character string consisting of two character strings composed of characters drawn from character set 01134. The two character strings are concatenated together by a period (the period is not part of character set 01134). The left-hand character string represents the network ID, and the right-hand character string represents the network LU name. The period is not part of the network ID or the network LU name. Neither network ID nor network LU name may be longer than eight bytes.

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

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

   The OS/400 CPI Communications support allows special values for the *partner_LU_name* conversation characteristic. While these special values may be specified only in the side information, they may be extracted with the Extract_Partner_LU_Name call if it is issued before the Allocate call. They cannot be used on the Set_Partner_LU_Name call. Refer to Chapter 11, “CPI Communications on Operating System/400” on page 513 for information about using these special values.

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

   - The space character is not allowed in *TP_name*.

- When communicating with non-CPI Communications programs, the *TP_name* can use characters other than those in character set 00640. See Appendix D, “CPI Communications and LU 6.2” on page 725 and “SNA Service Transaction Programs” on page 727 for details.
- On IMS and MVS systems, IBM recommends that the asterisk (*) be avoided in TP names because it causes a list request when it is entered on panels of the APPC/MVS administration dialog. The comma should also be avoided in IMS and MVS TP names, because it acts as a parameter delimiter in DISPLAY APPC commands.
- The OS/2 implementation allows the use of characters outside character set 01134 for the *TP_name* variable. See Chapter 10, “CPI Communications on OS/2” on page 435 for more information about default TP names used with OS/2.
- For OS/2 or Networking Services for Windows, the program can use the Set_TP_Name call to set the *TP_name* characteristic to an application TP name only. It cannot set the *TP_name* to an SNA service TP name because the encoding of the first character of an SNA service TP name overlaps with, and is indistinguishable from, some ASCII characters. In order to allocate a conversation with a partner SNA service TP, the SNA service TP name must be defined in the side information entry for the conversation.
- Networking Services for Windows translates TP names from ASCII to EBCDIC. Therefore, if the *TP_name* characteristic is set using a double-byte name and then the Allocate (CMALLC) call is issued, the partner LU will reject the allocation request because of an invalid TP name.

7. The field containing the *sym_dest_name* parameter on the CMINIT call must be eight bytes long. The symbolic destination name within that field may be from 0 to 8 characters long, with its characters taken from character set 01134. If the symbolic destination name is shorter than eight characters, it should be left-justified in the variable field, and padded on the right with spaces. A *sym_dest_name* parameter composed of eight spaces has special significance. See “Initialize_Conversation (CMINIT)” on page 200 for more information.
8. The four names in the following list are mode names defined by the LU 6.2 architecture for user sessions and may be specified for CPI Communications conversations on systems where they are defined, even though they contain the character **#**, which is not found in character set 01134:
   - #BATCH
   - #BATCHSC
   - #INTER
   - #INTERSC
9. The maximum size of the the *partner_ID* characteristic is determined by the *partner_ID_type*.
   - If the *partner_ID_type* is CM_PROGRAM_BINDING, the maximum size is 32767 bytes.
   - If the *partner_ID_type* is CM_DISTINGUISHED_NAME, CM_LOCAL_DISTINGUISHED_NAME, CM_PROGRAM_FUNCTION_ID, or CM_OSI_TPSU_TITLE_OID, the maximum size is 1024 bytes.

10. The *request_to_send_received* variable is used by CPI-C 1.2 programs. CPI-C 2.0 programs use *control_information_received*, which is an enhanced version of this variable.
11. The *callback_function* specifies a pointer to a routine and is supported by the C programming language only. Its length depends on the C compiler.

Distinguished Name

In general, directory objects are identified by a name. Different directories specify different syntax conventions for directory object names. Regardless of syntactic differences though, two naming convention commonalities can be identified:

- The program specifies a fully-qualified name that is valid for the entire distributed directory and namespace. A fully-qualified name may also be referred to as a global name.
- The program specifies an incomplete name that is only a part of a global name. An incomplete name is also referred to as a partial name. The global name can be created from a partial name by appending a prefix.

Here are some examples of global names using two different naming conventions. The first is an XDS-compliant distinguished name. The second is a name for the same object that would be recognized by DCE:

```
XDS:
     /C=us/O=UNC/OU=ChapelHill/CN=ThomasWolfe
DCE:
     /.../C=us/O=UNC/OU=ChapelHill/CN=ThomasWolfe
```

The distinguished name consists of a sequence of **relative distinguished names** (RDNs) separated by slashes. Each RDN is composed of an identifier that is set equal to a value. In the example, well-known abbreviations for identifiers are used—C for country, O for organization, OU for organization unit, CN for common name. Thus, the example name identifies an object for someone with a common name of ThomasWolfe at the Chapel Hill campus of the University of North Carolina (UNC). The "/.../" prefix at the front of the DCE name explicitly identifies the name as global.

The partial name of ThomasWolfe might be used to identify the object, but a prefix of /C=us/O=UNC/OU=ChapelHill/ would be required to make sure that it didn't locate a Tom Wolfe at the Greensboro campus of UNC. The specific DCE syntax for such a partial name would be:

```
/.:/ThomasWolfe
```

In DCE, the "/.:" prefix explicitly identifies the name as a partial distinguished name. The default prefix used by the local system to create a complete DN is the DCE "cell" name.

In CPI Communications, a distinguished name is always a global name. Programs establish a distinguished name by issuing the Set_Partner_ID call with a *partner_ID_type* set to CM_DISTINGUISHED_NAME. Programs establish a partial distinguished name by issuing the Set_Partner_ID call with a *partner_ID_type* set to CM_LOCAL_DISTINGUISHED_NAME. Format and syntax of the *partner_ID* characteristic are determined by the *directory_syntax* and *directory_encoding* characteristics.

Program Function Identifier (PFID)

The program function identifier (PFID) identifies the function provided by the program, thus allowing multiple installations of a given program to be recognized as providing the same function. This relationship is illustrated in Figure 30. In this example, Program A, B, and C all provide the same function. For example, they might be mail servers located on three different nodes. Conversations are allocated to each program using different destination information, and each would be identified with a different sym_dest_name or DN. The PFID allows all three programs to be identified as providing the same function.

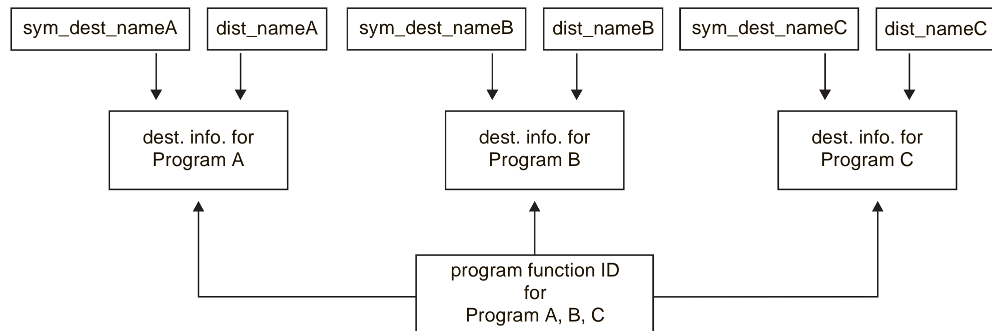


Figure 30. Relationship of PFID to Program Installation DNs

PFIDs must be administered to ensure they are unambiguous and no PFID value is assigned to more than one function. Unambiguous PFIDs allow use (and publication) of the PFID prior to creation of directory objects, and remove the requirement for installation questions about specific DNs. For example, code making use of PFIDs allows distribution of shrink-wrapped applications.

PFID Assignment Algorithms

CPI Communications does not establish any assignment algorithms or syntax rules for the PFID. It treats the PFID as a string and will retrieve program bindings by finding program installation objects that contain a matching PFID string. Assignment of PFIDs is implementation- and application- dependent.

Here are three sample assignment algorithms that ensure PFID uniqueness:

1. PFID values are Universal Unique Identifiers (UUIDs).
2. PFID values are ISO-registered object identifiers (OIDs).
3. A general-purpose naming method is created using existing naming registeries. Here is such a method:
   - The value consists of up to 1024 printable characters in GCSGID 640 (the set of invariant graphics in most character sets and also found in the invariant part of ISO 646).
   - The value is organized as set of three or more tokens separated by slashes(/).

- The first token identifies the existing naming authority. To accommodate the spectrum of developers, three values are proposed to begin with: “SNA,” “INTERNET,” and “OSI.”
- The second token is a unique value assigned by the naming authority identified in the first token. The token values vary based on the naming authority:
  - For SNA, the second token is a registered SNA NETID. For those network owners with more than one registered NETID, any one of them may be selected for this purpose. The third and last token is administered by the “owner” of the NETID.
  - For INTERNET, the second token is a printable form of a class A, B, or C NETID in decimal octet format (such as "9.0.0.0" for IBM). Alternatively, a domain name suffix such as “AUSTIN.IBM.COM” could be used. The third and last token is administered by the “owner” of the NETID.
  - For OSI the second token is a two character country code as described in ISO 3166 (such as US). The country code identifies the country under which the third token is registered. The third token is an organization code as registered by the registration authority in the country identified by the second token. (In the US, American National Standards Institute (ANSI) operates a registry for organizations.). The fourth token is administered by the organization. For example, if fictitious organization MYCOMPANY registered with ANSI, and identified a program function as ACCOUNTS_PAYABLE, the PFID would be:

```
/OSI/US/MYCOMPANY/ACCOUNTS_PAYABLE
```

Program Binding

A **program binding** contains the destination information required to allocate a conversation. The data in the binding is of the general pattern "field_type=field_value; field_type=field_value" where field_type is a four-byte identifier for the field and field_value is a variable-length field. The end of a field_value is determined by the semicolon character. If the field_value itself contains a semicolon, the textual semicolon is repeated.

Table 62 shows the 4-byte code for each field_type, along with a maximum length for each field_value. Note that the field_type of "BIND" has a maximum length of 0; this is because the BIND field_type is used to begin a binding and is immediately followed by another field_type. The binding is complete when either the end of the string is reached or another BIND field_type is reached.

Table 62. Fields in the Program Binding

Description of Field Contents	4-Byte field_type Code	Order	Maximum Length of Field
binding	BIND	1	0
CRM type	CRMT	2	5
TP name	TPNM	3	64
mode name	MODE	4	8
partner LU name	PLUN	5	17
partner principal name	PPNM	6	1024
required user name type	RUNM	7	9
AE qualifier	AEQL	8	1024
AP title	APTI	9	1024
application context name	APCN	10	256

A valid binding must contain both a CRM_type and TP_name field. The possible values of the CRM_type field_value are "LU6.2" and "OSITP." If the CRM_type is "LU6.2," the AE qualifier, AP title, and application context name fields should not be present. Similiarly, if the CRM_type is "OSITP," the partner LU name field should not be included in the program binding. Possible values for required user name type are: "NONE," "LOCAL," and "PRINCIPAL." The valid ordering of field_types for a single binding is as shown in Table 62.

Errors in the binding format will result in failure of the Allocate call with a return code of CM_PARAMETER_ERROR.

Note:

- Programs extracting a program binding after accepting a conversation may not receive a valid program binding because some fields may not be available to CPI Communications. Missing fields will have an field_type, but no field_value. For example, "TPNM=;" would be returned in a program binding for a program accepting a conversation with a CRM_type of "LU6.2." See "Extract_Partner_ID (CMEPID)" for further information.
- On systems using distributed directories that support multiple values for an attribute, the program_binding should be stored as a single attribute value.

Figure 31 shows a sample program binding for a program with the following destination information fields:

- CRM type of “LU6.2”
- TP name of “PAYROLL”
- Mode name of “BATCH”
- Partner LU name of “ibmnet07.accntlu0”

```
BIND=;CRMT=LU6.2;TPNM=PAYROLL;MODE=BATCH;PLUN=IBMNET07.ACCNTLU0;
```

Figure 31. Sample Program Binding Format

If the program can also be accessed by a second LU, that information can be included in the example by adding a “BIND=;” tag after the last semicolon and providing the next set of binding information.

Appendix B. Return Codes and Secondary Information

This chapter discusses the parameter called *return_code* that is passed back to the program at the completion of a call. It also discusses associated secondary information that may be available for the program to extract using the Extract_Secondary_Information call.

Return Codes

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

Some of the return codes indicate the results of the local processing of a call. These return codes are returned on the call that invoked the local processing. Other return codes indicate results of processing invoked at the remote end of the conversation. Depending on the call, these return codes can be returned on the call that invoked the remote processing or on a subsequent call. Still other return codes report events that originate at the remote end of the conversation. In all cases, only one code is returned at a time.

Some of the return codes associated with the allocation of a conversation have the suffix RETRY or NO_RETRY in their name.

- RETRY means that the condition indicated by the return code may not be permanent, and the program can try to allocate the conversation again. Whether or not the retry attempt succeeds depends on the duration of the condition. In general, the program should limit the number of times it attempts to retry without success.
- NO_RETRY means that the condition is probably permanent. In general, a program should not attempt to allocate the conversation again until the condition is corrected.

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

return_code >= CM_TAKE_BACKOUT

The return codes shown below are listed alphabetically, and each description includes the following:

- Meaning of the return code
- Origin of the condition indicated by the return code
- When the return code is reported to the program
- The state of the conversation when control is returned to the program

© Copyright IBM Corp. 1996, 1998

Notes:

1. The individual call descriptions in Chapter 4, “Call Reference” list the return code values that are valid for each call.
2. The integer values that correspond to the pseudonyms listed below are provided in Table 59 on page 642 of Appendix A, “Variables and Characteristics.”

The valid *return_code* values are described below:

CM_ALLOCATE_FAILURE_NO_RETRY
: The conversation cannot be allocated on a logical connection because of a condition that is not temporary. When this *return_code* value is returned to the program, the conversation is in **Reset** state. For example, if the conversation is using an LU 6.2 CRM, the logical connection (session) to be used for the conversation cannot be activated because the current session limit for the specified LU-name and mode-name pair is 0, or because of a system definition error or a session-activation protocol error. This return code is also returned when the session is deactivated because of a session protocol error before the conversation can be allocated. The program should not retry the allocation request until the condition is corrected. This return code is returned on the Allocate call.

CM_ALLOCATE_FAILURE_RETRY
: The conversation cannot be allocated on a logical connection because of a condition that may be temporary. When this *return_code* value is returned to the program, the conversation is in **Reset** state. For example, the logical connection to be used for the conversation cannot be activated because of a temporary lack of resources at the local system or remote system. This return code is also returned if the logical connection is deactivated because of logical connection outage before the conversation can be allocated. The program can retry the allocation request. This return code is returned on the Allocate call.

CM_ALLOCATION_ERROR
: This may be returned on calls associated with the Send queue (except the Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND) while the conversation is in **Send-Receive** state. The function requested on the call is not performed.

The return code indicates that the partner system rejected the conversation startup request. At the time this return code information is returned, the cause of allocation rejection is not returned to the program. The cause of the allocation rejection, which can be one of the following, can be obtained through the return code on the first Receive call.

- CM_SEND_RCV_MODE_NOT_SUPPORTED (OSI TP CRM only)
- CM_CONVERSATION_TYPE_MISMATCH (LU 6.2 CRM only)
- CM_PIP_NOT_SPECIFIED_CORRECTLY (LU 6.2 CRM only)
- CM_SECURITY_NOT_VALID (LU 6.2 CRM only)
- CM_SYNC_LEVEL_NOT_SUPPORTED_SYS (OSI TP CRM only)
- CM_SYNC_LEVEL_NOT_SUPPORTED_PGM (LU 6.2 CRM only)
- CM_TPN_NOT_RECOGNIZED
- CM_TP_NOT_AVAILABLE_NO_RETRY
- CM_TP_NOT_AVAILABLE_RETRY

The conversation is in **Receive-Only** state.

CM_BUFFER_TOO_SMALL
The local program issued a CPI Communications call specifying a buffer size that is insufficient for the amount of data available for the program to receive. The state of the conversation remains unchanged.

CM_CALL_NOT_SUPPORTED
The call is not supported by the local system. This return code is returned on any call in an optional conformance class when the implementation provides an entry point for the call but does not support the function requested by the call. The state of the conversation remains unchanged.

CM_CONV_DEALLOC_AFTER_SYNCPT (LU 6.2 CRM ONLY)
The conversation was deallocated as a part of the last sync-point operation. The local program was not given prior notification of the imminent deallocation because of a commit operation race that arose as follows: This program issued the resource recovery commit call. At the same time, the partner program issued a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* set to CM_SYNC_POINT_NO_CONFIRM, followed by the commit call.

CM_CONVERSATION_CANCELLED
This return code is returned for outstanding operations when the conversation was terminated locally by a Cancel_Conversation call.

CM_CONVERSATION_ENDING (LU 6.2 CRM ONLY)
This return code is returned on the Send_Expedited_Data and Receive_Expedited_Data calls and indicates one of the following:

- The local CRM is ending the conversation normally.
- A notification indicating that the remote program is ending the conversation (normally or abnormally) has been received by the local CRM.
- A notification of an error that causes the conversation to terminate has been received from the remote CRM or occurred locally.

The error that causes the conversation to terminate may be an allocation error, a conversation failure, or a deallocation of the conversation. The return code indicating that the cause of termination is returned on the calls associated with Send-Receive queue (half-duplex conversations only) or with the Send and Receive queues (full-duplex conversations only). The state of the conversation remains unchanged. Subsequent calls associated with the expedited queues will be rejected with this return code until the conversation enters **Reset** state.

CM_CONVERSATION_TYPE_MISMATCH (LU 6.2 CRM ONLY)
The remote system rejected the conversation startup request because of one of the following:

- The local program issued an Allocate call with *conversation_type* set to either CM_MAPPED_CONVERSATION or CM_BASIC_CONVERSATION, and the remote program does not support the respective conversation type.
- The local program issued an Allocate call with *send_receive_mode* set to either CM_HALF_DUPLEX or CM_FULL_DUPLEX, and the remote program does not support the respective send-receive mode.

For a half-duplex conversation, this return code is returned on a subsequent call to the Allocate. For a full-duplex conversation, this return code is returned on the Receive call. Calls associated with the Send queue that complete before this return code is returned on the Receive call are notified of the conversation type mismatch by a CM_ALLOCATION_ERROR return code.

When this *return_code* value is returned to the program, the conversation is in **Reset** state.

If *conversation_security_type* is set to CM_SECURITY_MUTUAL, then this return code may be returned on the Allocate call.

CM_DEALLOCATE_CONFIRM_REJECT (OSI TP CRM ONLY)
This return code is returned on a full-duplex conversation under one of the following two conditions:

- The program issued a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_CONFIRM and the partner program responded negatively to the Deallocate by issuing a Send_Error call in **Confirm-Deallocate** state.
- The program issued a Send_Data call with *send_type* set to CM_SEND_AND_DEALLOCATE and *deallocate_type* set to CM_DEALLOCATE_CONFIRM, and the partner program responded negatively to the Send-Data by issuing a Send_Error call in **Confirm-Deallocate** state.

A Receive call issued by the local program will receive a CM_PROGRAM_ERROR_PURGING return code after all available data is received. The state of the conversation remains unchanged.

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

- The remote program issued a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND, or a Cancel_Conversation call, or the remote system has done so because of a remote program abnormal-ending condition. If the remote program was in **Receive** state (half-duplex conversations only) or in **Send-Receive** or **Receive-Only** state (full-duplex conversations only) when the call was issued, information sent by the local program and not yet received by the remote program is purged.
- The remote program terminated normally but did not deallocate the conversation before terminating. Node services at the remote system deallocated the conversation on behalf of the remote program.
- On a half-duplex conversation using an OSI TP CRM, the local program issued a Send_Error call, and the error notification was delivered to the remote CRM. Subsequently, the remote program issued a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_FLUSH, or with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* set to CM_NONE, or with *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* set to CM_SYNC_POINT_NO_CONFIRM, and the conversation currently not included in a transaction.
- begin_transaction_collision (OSI TP CRM only)
  On a full-duplex conversation, there was a collision between a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_CONFIRM issued by the local program and an Include_Partner_In_Transaction call issued by the partner program. No log data is available.
- dealloc_confirm_collision (OSI TP CRM only)
  On a full-duplex conversation, there was a collision between a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_CONFIRM issued by the local program and a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_CONFIRM call issued by the partner program. No log data is available.

- On a full-duplex conversation in **Send-Receive** state or **Receive-Only** state, the local program has issued a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND.
- CPI Communications deallocated the conversation because an implicit call of tx_set_transaction_control or tx_begin failed.

For a half-duplex conversation, this return code is reported to the local program on a call issued in **Send** or **Receive** state. For a full-duplex conversation, this return code is returned on a Receive call issued in **Send-Receive** or **Receive-Only** state. It is also returned on calls associated with the Send queue (except the Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND) under one of the following conditions:

- They are issued in **Send-Only** state.
- They are issued in **Send-Receive** state and complete before this return code is returned on the Receive call.

The conversation is now in **Reset** state unless the return code was returned on one of the calls associated with the Send queue, issued in **Send-Receive** state. In that case, the conversation is in **Receive-Only** state.

CM_DEALLOCATED_ABEND_BO

This return code is returned only for conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and with the conversation included in a transaction.

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

- The remote program issued a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND or a Cancel_Conversation call, or the remote system has done so because of a remote program abnormal-ending condition.
- dealloc_cfm_collision_bo (OSI TP CRM only)
  On a full-duplex conversation, there was a collision between a Include_Partner_In_Transaction call issued by this program and a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_CONFIRM issued by the partner program. The Include_Partner_In_Transaction call got a return code of CM_OK. However, when the collision is detected, a CM_DEALLOCATED_ABEND_BO return code is returned on a subsequent call. No log data is available.

If the remote program was in **Receive** state (half-duplex conversations only) or in **Send-Receive**, **Prepared** or **Deferred-Deallocate** states (full-duplex conversations only) when it issued the Deallocate call, information sent by the local program and not yet received by the remote program is purged.

For a half-duplex conversation, this return code is reported to the local program on a call issued in **Send** or **Receive** state. For a full-duplex conversation, this return code is reported to the local program on calls issued in **Send-Receive**, **Sync-Point**, **Deferred-Deallocate**, **Sync-Point-Deallocate** and **Prepared** states. The conversation is now in **Reset** state. For a full-duplex conversation, incoming information may not be received if this return code is returned on a call associated with the Send queue.

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

CM_DEALLOCATED_ABEND_SVC (LU 6.2 CRM ONLY)
This return code is returned for basic conversations only. It may be returned under one of the following conditions:

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

For a half-duplex conversation, this return code is reported to the local program on a call issued in **Send** or **Receive** state. For a full-duplex conversation, this return code is returned on a Receive call issued in **Send-Receive** or **Receive-Only** state. It is also returned on calls associated with the Send queue (except the Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND) under one of the following conditions:

- They are issued in **Send-Only** state.
- They are issued in **Send-Receive** state and complete before this return code is returned on the Receive call.

The conversation is now in **Reset** state unless the return code was returned on one of the calls associated with the Send queue issued in **Send-Receive** state. In that case, the conversation is in **Receive-Only** state.

CM_DEALLOCATED_ABEND_SVC_BO (LU 6.2 CRM ONLY)
This return code is returned only for basic conversations with *sync_level* set to CM_SYNC_POINT. It is returned under the same conditions described under CM_DEALLOCATED_ABEND_SVC above.

For a half-duplex conversation, this return code is reported to the local program on a call issued in **Send** or **Receive** state. For a full-duplex conversation, this return code is reported to the local program on calls issued in **Send-Receive**, **Sync-Point**, **Deferred-Deallocate**, **Sync-Point-Deallocate**, and **Prepared** states. The conversation is now in **Reset** state.

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

CM_DEALLOCATED_ABEND_TIMER (LU 6.2 CRM ONLY)
This return code is returned only for basic conversations.

In addition, it is returned only when the remote program is using an LU 6.2 application programming interface and is not using CPI Communications.

The remote LU 6.2 transaction program issued a DEALLOCATE verb specifying a TYPE parameter of ABEND_TIMER. For a half-duplex conversation, this return code is reported to the local program on a call issued in **Send** or **Receive** state. For a full-duplex conversation, this return code is returned on a Receive call issued in **Send-Receive** or **Receive-Only** state.

It is also returned on calls associated with the Send queue (except the Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND) under one of the following conditions:

- They are issued in **Send-Only** state.
- They are issued in **Send-Receive** state and complete before this return code is returned on the Receive call.

The conversation is now in **Reset** state unless the return code was returned on one of the calls associated with the Send queue issued in **Send-Receive** state. In that case, the conversation is in **Receive-Only** state.

CM_DEALLOCATED_ABEND_TIMER_BO (LU 6.2 CRM ONLY)

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

The remote LU 6.2 transaction program issued a DEALLOCATE verb specifying a TYPE parameter of ABEND_TIMER. If the conversation for the remote program was in **Receive** state (half-duplex conversations only) or in **Send-Receive**, **Prepared**, or **Deferred-Deallocate** state (full-duplex conversations only) when the verb was issued, information sent by the local program and not yet received by the remote program is purged. If the return code is returned on a call associated with the Send queue for a full-duplex conversation, incoming data may be purged. For a half-duplex conversation, this return code is reported to the local program on a call issued in **Send** or **Receive** state. For a full-duplex conversation, this return code is reported to the local program on calls issued in **Send-Receive**, **Sync-Point**, **Deferred-Deallocate**, **Sync-Point-Deallocate**, and **Prepared** states. The conversation is now in **Reset** state.

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

CM_DEALLOCATED_NORMAL

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

- The remote program issued a Deallocate call or a Send_Data call with *send_type* set to CM_SEND_AND_DEALLOCATE on a basic or mapped conversation with one of the following:
  - *deallocate_type* set to CM_DEALLOCATE_FLUSH
  - *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* set to CM_NONE
  - *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* set to CM_SYNC_POINT_NO_CONFIRM, and the conversation is not currently included in a transaction

  For a half-duplex conversation, this return code is reported to the local program on a call issued in **Receive** state. For a full-duplex conversation, this return code is reported to the local program on the Receive call issued in **Send-Receive** or **Receive-Only** state. If the conversation is a full-duplex conversation using an OSI TP CRM, this return code is also returned on calls associated with the Send queue (except the Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND).

- The local program issued a Deallocate call or a Send_Data call with *send_type* set to CM_SEND_AND_DEALLOCATE and with one of the following:
  - *deallocate_type* set to CM_DEALLOCATE_FLUSH
  - *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* set to CM_NONE, and the conversation is a full-duplex conversation using an OSI TP CRM
  - *deallocate_type* set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* set to CM_SYNC_POINT_NO_CONFIRM, and the conversation is not currently included in a transaction

  This return code is returned to the local program on a Receive call that was outstanding when the Deallocate call was issued.

For a half-duplex conversation, the conversation is now in **Reset** state. For a full-duplex conversation, the conversation can now be in one of the following states:

- **Reset** state if this return code was returned on calls issued in **Receive-Only** or **Send-Only** state.
- **Send-Only** state if this return code was returned on a Receive call issued in **Send-Receive** state.
- **Receive-Only** state if this return code was returned on calls associated with the Send queue (except the Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND) issued in **Send-Receive** state.

CM_DEALLOCATED_NORMAL_BO

This return code is returned only for half-duplex conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and with the conversation included in a transaction.

When the conversation is using an LU 6.2 CRM and the Send_Error call is issued in **Receive** state, incoming information is purged by the system. This purged information may include an abend deallocation notification from the remote program or system. When such a notification is purged, CPI Communications returns CM_DEALLOCATED_NORMAL_BO instead of one of the following return codes:

- CM_DEALLOCATED_ABEND_BO
- CM_DEALLOCATED_ABEND_SVC_BO
- CM_DEALLOCATED_ABEND_TIMER_BO

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

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

CM_EXP_DATA_NOT_SUPPORTED (LU 6.2 CRM ONLY)

An expedited data call was locally rejected because the remote CRM does not support expedited data. The state of the conversation remains unchanged.

CM_INCLUDE_PARTNER_REJECT_BO (OSI TP CRM ONLY)

A prior Include_Partner_In_Transaction call issued by the program completed locally with CM_OK but was rejected by the partner system. The partner system rejected the request to join the transaction because the partner program is already a part of another transaction and cannot be a part of two transactions at the same time. The conversation is now in **Reset** state.

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

CM_MAP_ROUTINE_ERROR
: This return code is returned on Send_Mapped_Data, Receive_Mapped_Data, Extract_Mapped_Initialization_Data, and Set_Mapped_Initialization_Data, indicating that the underlying map routine aborted. If the error occurred during a receive function, the incoming data is placed in the buffer un-decoded so the application program can take appropriate action.

CM_NO_SECONDARY_INFORMATION
: The Extract_Secondary_Information call did not complete successfully because no secondary information was available for the specified call on the specified conversation. The state of the conversation remains unchanged.

CM_OK
: The call issued by the local program executed successfully (that is, the function defined for the call, up to the point at which control is returned to the program, was performed as specified). The state of the conversation is as defined for the call.

CM_OPERATION_INCOMPLETE
: A non-blocking operation has been started either on the conversation (when conversation-level non-blocking is used) or on the queue with which the call is associated (when queue-level non-blocking is used), but the operation has not completed. This return code is returned when the call is suspended waiting for incoming data, buffers, or other resources. A program must do one of the following:

- For conversation-level non-blocking, use the Wait_For_Conversation call to wait for the operation to complete and to retrieve the return code for the completed operation.
- For queue-level non-blocking,
  - If an OOID is associated with the outstanding operation, use the Wait_For_Completion call to wait for the operation to complete and to obtain the OOID and user field corresponding to the completed operation.
  - If a callback function is associated with the outstanding operation, use the callback function and user field to properly handle the completed operation.

The state of the conversation remains unchanged.

CM_OPERATION_NOT_ACCEPTED
: A previous operation either on this conversation (when conversation-level non-blocking is chosen) or on the same queue (when conversation-level non-blocking is not chosen) is incomplete. This return code is returned when there is an outstanding operation on the conversation or queue, as indicated by the CM_OPERATION_INCOMPLETE return code to a previous call. On a system that supports multiple program threads, when one thread has started an operation that has not completed, this return code is returned on a call made by another thread on the same conversation or associated with the same queue. The state of the conversation remains unchanged.

CM_PARM_VALUE_NOT_SUPPORTED
The specified value of a call parameter is not supported by the local system. This return code is returned on a call with defined parameter values that are optional for support of the call. It is returned when the implementation supports the call but does not support the specified optional parameter value. The state of the conversation remains unchanged.

CM_PARAMETER_ERROR
The local program issued a call specifying a parameter containing an invalid argument. ("Parameters" include not only the parameters described as part of the call syntax, but also characteristics associated with the *conversation_ID*.) The source of the argument is considered to be outside the program definition, such as an LU name supplied by a system administrator in the side information and referenced by the Initialize_Conversation call.

The CM_PARAMETER_ERROR return code is returned on the call specifying the invalid argument. The state of the conversation remains unchanged.

Note: Contrast this definition with the definition of the CM_PROGRAM_PARAMETER_CHECK return code.

CM_PIP_NOT_SPECIFIED_CORRECTLY (LU 6.2 CRM ONLY)
This return code is returned only when the remote program is using an LU 6.2 application programming interface and is not using CPI Communications.

The remote CRM rejected the conversation startup request because the remote program has one or more program initialization parameter (PIP) variables defined and the initialization data specified by the local program is incorrect. This return code is returned on a call issued after the Allocate for a half-duplex conversation. For a full-duplex conversation, this return code is returned on the Receive call. Calls associated with the Send queue that complete before this return code is returned are notified of the error by an CM_ALLOCATION_ERROR return code. When this return code is returned to the program, the conversation is in **Reset** state.

If *conversation_security_type* is set to CM_SECURITY_MUTUAL, then this return code may be returned on the Allocate call.

CM_PRODUCT_SPECIFIC_ERROR
A product-specific error has been detected and a description of the error has been entered into the product's system error log. See product documentation for an indication of conditions and state changes caused by this return code.

CM_PROGRAM_ERROR_NO_TRUNC (LU 6.2 CRM ONLY)
One of the following occurred:

- The remote program issued a Send_Error call on a mapped conversation and the conversation for the remote program was in **Send** state (half-duplex conversations only) or in **Send-Receive** or **Send-Only** state (full-duplex conversations only). No truncation occurs at the mapped conversation protocol boundary. This return code is reported to the local program on a Receive call the program issues before receiving any data records or after receiving one or more data records.
- The remote program issued a Send_Error call on a basic conversation, the conversation for the remote program was in **Send** state (half-duplex conversations only) or in **Send-Receive** or **Send-Only** state (full-duplex conversations only), and the call did not truncate a logical record. No truncation occurs at the basic conversation protocol boundary when a program issues Send_Error before sending any logical records or after

sending a complete logical record. This return code is reported to the local program on a Receive call the program issues before receiving any logical records or after receiving one or more complete logical records.

- The remote program issued a Send_Error call on a mapped or basic half-duplex conversation and the conversation for the remote program was in **Send-Pending** state. No truncation of data has occurred. This return code indicates that the remote program has issued Set_Error_Direction to set the *error_direction* characteristic to CM_SEND_ERROR. The return code is reported to the local program on a Receive call the program issues before receiving any data records or after receiving one or more data records.

The conversation remains in **Receive** state for a half-duplex conversation or in **Send-Receive** or **Receive-Only** state for a full-duplex conversation.

CM_PROGRAM_ERROR_PURGING
One of the following occurred:

- The remote program issued a Send_Error call on a basic or mapped half-duplex conversation while its end of the conversation was in **Receive** or **Confirm** state. The call may have caused information enroute to the remote program to be purged (discarded), but not necessarily.

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

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

- The remote program issued a Send_Error call on a mapped or basic half-duplex conversation and the conversation for the remote program was in **Send-Pending** state. No purging of data has occurred. This return code indicates that the remote program has issued a Send_Error call with *error_direction* set to CM_RECEIVE_ERROR.

- The full-duplex conversation is allocated using an OSI TP CRM and the remote program issued a Send_Error call while its end of the conversation was in **Send-Receive**, **Send-Only**, or **Confirm-Deallocate** state. Purging of data sent by the local program may have occurred in transit, if the remote program was in **Send-Receive** or **Send-Only** state when it issued Send_Error.

For a half-duplex conversation, this return code is normally reported to the local program on a call the program issues after sending some information to the remote program. However, the return code can be reported on a call the program issues before sending any information, depending on the call and when it is issued. For a full-duplex conversation, this return code is returned on the Receive call. The half-duplex conversation remains in **Receive** state. The full-duplex conversation remains in **Send-Receive** or **Receive-Only** state.

CM_PROGRAM_ERROR_TRUNC (LU 6.2 CRM ONLY)

The remote program issued a Send_Error call on a basic conversation, the conversation for the remote program was in **Send** state (half-duplex conversations only) or in **Send-Receive** or **Send-Only** state (full-duplex conversations only), and the call truncated a logical record. Truncation occurs at the basic conversation protocol boundary when a program begins sending a logical record and then issues Send_Error before sending the complete logical record. This return code is reported to the local program on a Receive call the program issues after receiving the truncated logical record. The conversation remains in **Receive** state for a half-duplex conversation or in **Send-Receive** or **Receive-Only** state for a full-duplex conversation.

CM_PROGRAM_PARAMETER_CHECK

The local program issued a call in which a programming error has been found in one or more parameters. ("Parameters" include not only the parameters described as part of the call syntax, but also characteristics associated with the *conversation_ID*, the CRM type used by the conversation, and the transaction role (superior or subordinate) of the program.) The source of the error is considered to be inside the program definition (under the control of the local program). This return code may be caused by the failure of the program to pass a valid parameter address. The program should not examine any other returned variables associated with the call as nothing is placed in the variables. The state of the conversation remains unchanged.

CM_PROGRAM_STATE_CHECK

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

- The local program issued a call for a conversation in a state that was not valid for that call.
- There is no incoming conversation. The Accept_Conversation call was issued but did not complete successfully.
- No name is associated with the program. The Accept_Conversation or Accept_Incoming call was issued but did not complete successfully.
- The program started but did not finish sending a logical record.
- There is no outstanding operation. The Wait_For_Completion or Wait_For_Conversation call was issued but did not complete successfully.
- For a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the conversation's context is in the **Backout-Required** condition. The program issued a call that is not allowed for this conversation while its context is in this condition.
- The program has received a *status_received* value of CM_JOIN_TRANSACTION. The program issued a call that is not allowed before the program joins the transaction.
- The conversation is included in a transaction. The program issued a call that is allowed only when the conversation is not currently included in a transaction.
- The conversation is not currently included in a transaction. The program issued a call that is allowed only when the conversation is included in a transaction.
- A prior Deferred_Deallocate call is still in effect for the conversation. The Prepare_To_Receive call was issued but is not allowed.

- The program has not received a take-commit notification from its superior. The Prepare call was issued but is not allowed.

The program should not examine any other returned variables associated with the call as nothing is placed in the variables. The state of the conversation remains unchanged.

CM_RESOURCE_FAILURE_NO_RETRY

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

- A failure occurred that caused the conversation to be prematurely terminated. For example, the logical connection being used for the conversation was deactivated because of a logical-connection protocol error, or the conversation was deallocated because of a protocol error between the mapped conversation components of the systems. The condition is not temporary, and the program should not retry the transaction until the condition is corrected.
- The remote program terminated normally but did not deallocate the conversation before terminating. Node services at the remote system deallocated the conversation on behalf of the remote program.

This return code can be reported to the local program on a call it issues for a conversation in any state other than **Reset** or **Initialize** state for a half-duplex or full-duplex conversation, or **Sync-Point**, **Sync-Point-Deallocate**, or **Defer-Deallocate** state for a full-duplex conversation.

If *conversation_security_type* is set to CM_SECURITY_MUTUAL, then this return code may be returned on the Allocate call.

For a full-duplex conversation, this return code is returned on the Receive call, at which time the conversation goes to **Reset** state. Calls associated with the Send queue (except the Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND) that complete before this return code is returned on the Receive call also get this return code, and the conversation is in **Receive-Only** or **Reset** state, depending on whether the call was issued in **Send-Receive** or **Send-Only** state. The conversation is in **Reset** state if this a half-duplex conversation.

CM_RESOURCE_FAIL_NO_RETRY_BO

This return code is returned only for conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and with the conversation included in a transaction.

A failure occurred that caused the conversation to be prematurely terminated. For example, the logical connection being used for the conversation was deactivated because of a logical-connection protocol error, or the conversation was deallocated because of a protocol error between the mapped conversation components of the systems. The condition is not temporary, and the program should not retry the transaction until the condition is corrected. This return code can be reported to the local program on a call issued in any state other than **Reset** or **Initialize** state. For a full-duplex conversation, incoming information may not be received if this return code is returned on a call associated with the Send queue. The conversation is in **Reset** state.

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

CM_RESOURCE_FAILURE_RETRY

A failure occurred that caused the conversation to be prematurely terminated. For example, the logical connection being used for the conversation was deactivated because of a logical-connection outage such as a line failure, a modem failure, or a crypto engine failure. The condition may be temporary, and the program can retry the transaction.

This return code can be reported to the local program on a call it issues for a conversation in any state other than **Reset** or **Initialize** state for a half-duplex or full-duplex conversation, or **Sync-Point**, **Sync-Point-Deallocate**, or **Defer-Deallocate** state for a full-duplex conversation. For a full-duplex conversation, this return code is returned on the Receive call, at which time the conversation goes to **Reset** state. Calls associated with the Send queue (except the Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND) that complete before this return code is returned on the Receive call also get this return code, and the conversation is in **Receive-Only** or **Reset** state, depending on whether the call was issued in **Send-Receive** or **Send-Only** state. The conversation is in **Reset** state if this is a half-duplex conversation.

If *conversation_security_type* is set to CM_SECURITY_MUTUAL, then this return code may be returned on the Allocate call.

CM_RESOURCE_FAILURE_RETRY_BO

This return code is returned only for conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and with the conversation included in a transaction.

A failure occurred that caused the conversation to be prematurely terminated. For example, the logical connection being used for the conversation was deactivated because of a logical-connection outage such as a line failure, a modem failure, or a crypto engine failure. The condition may be temporary, and the program can retry the transaction. This return code can be reported to the local program on a call it issues for a conversation in any state other than **Reset** or **Initialize**. For a full-duplex conversation, incoming information may not be received if this return code is returned on a call associated with the Send queue. The conversation is in **Reset** state.

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

CM_RETRY_LIMIT_EXCEEDED

The conversation cannot be allocated on a logical connection because CPI Communications has exceeded the local system's retry limit. When this *return_code* value is returned to the program, the conversation is in **Reset** state.

CM_SECURITY_MUTUAL_FAILED

This return code is returned only for conversations with *conversation_security_type* set to CM_SECURITY_MUTUAL.

The local system failed the allocate request because the local system was not able to authenticate the remote system. When this *return_code* value is returned to the program, the conversation is in **Reset** state.

CM_SECURITY_NOT_SUPPORTED

The local system rejected the allocate request because the local program specified a required user name type and conversation security type combination that is not supported between the local and remote systems. When this *return_code* value is returned to the program, the conversation is in **Reset** state.

CM_SECURITY_NOT_VALID

The remote system rejected the conversation startup request because the access security information (provided by the local system) is invalid. This return code is returned on a call issued after the Allocate for a half-duplex conversation. For a full-duplex conversation, this return code is returned on the Receive call. Calls associated with the Send queue that complete before this return code is returned on the Receive call are notified of the error by a CM_ALLOCATION_ERROR return code. When this *return_code* value is returned to the program, the conversation is in **Reset** state.

If *conversation_security_type* is set to CM_SECURITY_MUTUAL, then this return code may be returned on the Allocate call.

CM_SEND_RCV_MODE_NOT_SUPPORTED

This return code indicates that the conversation startup request was rejected because of one of the following:

- The *send_receive_mode* characteristic is set to CM_HALF_DUPLEX but the remote system does not support half-duplex conversations.
- The *send_receive_mode* characteristic is set to CM_FULL_DUPLEX but the remote system does not support full-duplex conversations.

The state of the conversation remains unchanged.

CM_SVC_ERROR_NO_TRUNC (LU 6.2 CRM ONLY)

This return code is returned only for basic conversations. In addition, it is returned only when the remote program is using an LU 6.2 application programming interface and is not using CPI Communications.

The remote LU 6.2 transaction program issued a Send_Error verb specifying a TYPE parameter of SVC, the conversation for the remote program was in **Send** state for a half-duplex conversation or in **Send-Receive** or **Send-Only** state for a full-duplex conversation, and the verb did not truncate a logical record. This return code is returned on a Receive call. When this return code is returned to the local program on a half-duplex conversation, the conversation is in **Receive** state. There is no state change for a full-duplex conversation.

CM_SVC_ERROR_PURGING (LU 6.2 CRM ONLY)

This return code is returned only for basic half-duplex conversations. In addition, it is returned only when the remote program is using an LU 6.2 application programming interface and is not using CPI Communications.

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

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

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

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

The local program specified a *sync_level* of CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, which the remote system does not support. This return code is returned on the Allocate call.

For a full-duplex conversation, this return code is returned on the Receive call if an attempt to allocate the conversation was made by the local program running on an OSI TP CRM and the remote system does not support the *sync_level* of CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM specified in the conversation startup request. The remote system has rejected the allocation attempt. Calls associated with the local Send queue that complete before this return code is returned on the Receive call are notified of the error by a CM_ALLOCATION_ERROR return code.

When this *return_code* value is returned to the program, the conversation is in **Reset** state.

CM_SYNC_LVL_NOT_SUPPORTED_PGM
The remote system rejected the conversation startup request because the local program specified a synchronization level (with the *sync_level* parameter) that the remote program does not support. For a half-duplex conversation, this return code is returned on a call issued after the Allocate. For a full-duplex conversation, this return code is returned on the Receive call. Calls associated with the Send queue that complete before this return code is returned on the Receive call are notified of the error by a CM_ALLOCATION_ERROR return code.

When this *return_code* value is returned to the program, the conversation is in **Reset** state.

If *conversation_security_type* is set to CM_SECURITY_MUTUAL, then this return code may be returned on the Allocate call.

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

CM_TAKE_BACKOUT

This return code is returned only for conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and with the conversation included in a transaction.

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

CM_TPN_NOT_RECOGNIZED

The remote system rejected the conversation startup request because the local program specified a remote program name that the remote system does not recognize. For a half-duplex conversation, this return code is returned on a call issued after the Allocate. For a full-duplex conversation, this return code is returned on the Receive call. Calls associated with the Send queue that complete before this return code is returned on the Receive call are notified of the error by a CM_ALLOCATION_ERROR return code. When this *return_code* value is returned to the program, the conversation is in **Reset** state.

If *conversation_security_type* is set to CM_SECURITY_MUTUAL, then this return code may be returned on the Allocate call.

CM_TP_NOT_AVAILABLE_NO_RETRY

The remote system rejected the conversation startup request because the local program specified a remote program that the remote system recognizes but cannot start. The condition is not temporary, and the program should not retry the allocation request. For a half-duplex conversation, this return code is returned on a call issued after the Allocate. For a full-duplex conversation, this return code is returned on the Receive call. Calls associated with the Send queue that complete before this return code is returned on the Receive call are notified of the error by a CM_ALLOCATION_ERROR return code. When this *return_code* value is returned to the program, the conversation is in **Reset** state.

If *conversation_security_type* is set to CM_SECURITY_MUTUAL, then this return code may be returned on the Allocate call.

CM_TP_NOT_AVAILABLE_RETRY

The remote system rejected the conversation startup request because the local program specified a remote program that the remote system recognizes but currently cannot start. The condition may be temporary, and the program can retry the allocation request. For a half-duplex conversation, this return code is returned on a call issued after the Allocate. For a full-duplex conversation, this return code is returned on the Receive call. Calls associated with the Send queue that complete before this return code is returned on the Receive call are notified of the error by a CM_ALLOCATION_ERROR return code. When this *return_code* value is returned to the program, the conversation is in **Reset** state.

If *conversation_security_type* is set to CM_SECURITY_MUTUAL, then this return code may be returned on the Allocate call.

CM_UNKNOWN_MAP_NAME_RECEIVED

This return code is returned on Receive_Mapped_Data or Extract_Mapped_Initialization_Data when the incoming map name is invalid

because the map routine cannot recognize it. The function request returns the undecoded incoming data in the buffer and expects the application program to take appropriate action. This is a serious error which indicates that the two partners do not have the necessary common understanding to communicate. Under the OSI TP CRM, this would indicate that one of the partners has violated the associations's application context. Unser the LU 6.2 CRM, both partners should review what is expected and determine which partner is in error.

CM_UNKNOWN_MAP_NAME_REQUESTED
: This return code is returned on Send_Mapped_Data, or Set_Mapped_Initialization_Data when a map_name is specified that the underlying map routine finds is invalid. The function requested is not performed and the application program can perform appropriate error recovery. A user should check to ensure that the proper map routine is being used and the spelling of the map_name.

CM_UNSUCCESSFUL
: The call issued by the local program did not execute successfully. This return code is returned on the unsuccessful call. The state of the conversation remains unchanged.

Secondary Information

Associated with the return code, there may be secondary information available for the program to extract using the Extract_Secondary_Information call. The secondary information can be used to determine the cause of the return code and to aid problem determination. Based on its origin, the secondary information and associated return code can belong to one of the four types, as shown in Table 63.

Table 63. Secondary Information Types and Associated Return Codes

Secondary Information Type	Associated Return Codes
Application-oriented	CM_DEALLOCATED_ABEND CM_DEALLOCATED_ABEND_BO CM_DEALLOCATED_ABEND_SVC CM_DEALLOCATED_ABEND_SVC_BO CM_DEALLOCATED_ABEND_TIMER CM_DEALLOCATED_ABEND_TIMER_BO CM_PROGRAM_ERROR_NO_TRUNC CM_PROGRAM_ERROR_PURGING CM_PROGRAM_ERROR_TRUNC CM_SVC_ERROR_NO_TRUNC CM_SVC_ERROR_PURGING CM_SVC_ERROR_TRUNC
CPI Communications-defined	CM_DEALLOCATED_ABEND CM_DEALLOCATED_ABEND_BO CM_PARAMETER_ERROR CM_PROGRAM_PARAMETER_CHECK CM_PROGRAM_STATE_CHECK CM_SECURITY_NOT_SUPPORTED
CRM-specific	CM_ALLOCATE_FAILURE_NO_RETRY CM_ALLOCATE_FAILURE_RETRY CM_CONVERSATION_TYPE_MISMATCH CM_PIP_NOT_SPECIFIED_CORRECTLY CM_RESOURCE_FAIL_NO_RETRY_BO CM_RESOURCE_FAILURE_NO_RETRY CM_RESOURCE_FAILURE_RETRY CM_RESOURCE_FAILURE_RETRY_BO CM_RETRY_LIMIT_EXCEEDED CM_SECURITY_MUTUAL_FAILED CM_SECURITY_NOT_SUPPORTED CM_SECURITY_NOT_VALID CM_SEND_RCV_MODE_NOT_SUPPORTED CM_SYNC_LVL_NOT_SUPPORTED_PGM CM_SYNC_LVL_NOT_SUPPORTED_SYS CM_TP_NOT_AVAILABLE_NO_RETRY CM_TP_NOT_AVAILABLE_RETRY CM_TPN_NOT_RECOGNIZED
Implementation-related	CM_PRODUCT_SPECIFIC_ERROR

The following return codes, upon being returned to the program, are not associated with any secondary information:

- CM_ALLOCATION_ERROR
- CM_BUFFER_TOO_SMALL
- CM_CALL_NOT_SUPPORTED
- CM_CONV_DEALLOC_AFTER_SYNCPT
- CM_CONVERSATION_CANCELLED
- CM_CONVERSATION_ENDING
- CM_DEALLOC_CONFIRM_REJECT
- CM_DEALLOCATED_NORMAL
- CM_DEALLOCATED_NORMAL_BO

- CM_EXP_DATA_NOT_SUPPORTED
- CM_INCLUDE_PARTNER_REJECT_BO
- CM_MAP_ROUTINE_ERROR
- CM_NO_SECONDARY_INFORMATION
- CM_OK
- CM_OPERATION_INCOMPLETE
- CM_OPERATION_NOT_ACCEPTED
- CM_PARM_VALUE_NOT_SUPPORTED
- CM_SYSTEM_EVENT
- CM_TAKE_BACKOUT
- CM_UNSUCCESSFUL
- CM_UNKNOWN_MAP_NAME_RECEIVED
- CM_UNKNOWN_MAP_NAME_REQUESTED

Except for application-oriented information, which is defined entirely by the application, secondary information is a string of printable characters and, in general, consists of the following information in the order described:

1. Condition code
2. Description of the condition
3. Cause of the condition
4. Suggested actions
5. Additional information from the implementation

For different secondary information types, the condition codes are in the range specified in Table 64. In some cases, secondary information may not have all these fields. Fields present in secondary information are separated by two consecutive semicolons. The following sections provide examples of secondary information in different types.

Table 64. Range of Condition Codes for Different Secondary Information Types

Secondary Information Type	**Condition Codes**
CPI Communications-defined	1 - 4000
CRM-specific	4001 (for an LU 6.2 CRM), 4002 (for an OSI TP CRM)
Implementation-related	4003

Application-Oriented Information

When a program discovers an abnormal condition during its processing, the program may use log data to convey the condition to its partner program. The partner program receives the log data when it issues the Extract_Secondary_Information call. Since log data is application data, it is up to the application designer to define and interpret its content.

CPI Communications-Defined Information

Table 65 lists all CPI Communications-defined secondary information.

Table 65 (Page 1 of 11). CPI Communications-Defined Secondary Information

Condition Code	Description
Associated with CM_PROGRAM_PARAMETER_CHECK:	0 < *n* < 101
The *n*th parameter specifies an invalid address.	101
The *conversation_ID* specifies an unassigned conversation identifier.	102
The *sync_level* is set to CM_NONE.	103
The *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM.	104
The *send_receive_mode* is set to CM_FULL_DUPLEX.	105
The *send_receive_mode* is set to CM_FULL_DUPLEX, and conversation is using an LU 6.2 CRM.	106
The *buffer_length* specifies a value that is invalid for the range permitted by the implementation.	107
The conversation is using an OSI TP CRM, and the program is not the superior for the conversation.	108
The conversation is using an LU 6.2 CRM.	109
The *requested_length* specifies a value less than 0.	110
The *transaction_control* is set to CM_CHAINED_TRANSACTIONS.	111
The *sym_dest_name* specifies an unrecognized value.	112
The *sync_level* is set to CM_CONFIRM.	113
The *requested_length* specifies a value that exceeds the range permitted by the implementation.	114
The *requested_length* specifies a value less than 0 or greater than 86.	115
The *expedited_receive_type* specifies an undefined value.	116
The conversation is using an OSI TP CRM.	117
The *TP_name* specifies a name that is not associated with this program.	118
The *TP_name_length* specifies a value less than 1 or greater than 64.	119
The *send_length* specifies a value that exceeds the range permitted by the implementation.	120
The *conversation_type* is set to CM_BASIC_CONVERSATION and *buffer* contains an invalid logical record length (LL) value of X'0000', X'0001', X'8000', X'8001'.	121
The *send_type* is set to CM_SEND_AND_PREP_TO_RECEIVE, *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT, the conversation using an OSI TP CRM is included in a transaction, and the program is not the superior for the conversation.	122

Table 65 (Page 2 of 11). CPI Communications-Defined Secondary Information

Condition Code	Description
The *send_type* is set to CM_SEND_AND_PREP_TO_RECEIVE, *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, the conversation using an OSI TP CRM is included in a transaction, and the program is not the superior for the conversation.	123
The *send_type* is set to CM_SEND_AND_DEALLOCATE, *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT, the conversation using an OSI TP CRM is included in a transaction, and the program is not the superior for the conversation.	124
The *send_type* is set to CM_SEND_AND_DEALLOCATE, *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, the conversation using an OSI TP CRM is included in a transaction, and the program is not the superior for the conversation.	125
The *send_length* specifies a value less than 1 or greater than 86.	126
The *AE_qualifier_length* specifies a value less than 1 or greater than 1024.	127
The *AE_qualifier_format* specifies an undefined value.	128
The *partner_ID* is set to a non-null value.	129
The *allocate_confirm* specifies an undefined value.	130
The *allocate_confirm* specifies CM_ALLOCATE_CONFIRM, and the conversation is using an LU 6.2 CRM.	131
The *AP_title_length* specifies a value less than 1 or greater than 1024.	132
The *AP_title_format* specifies an undefined value.	133
The *application_context_name_length* specifies a value less than 1 or greater than 256.	134
The *begin_transaction* specifies an undefined value.	135
The *confirmation_urgency* specifies an undefined value.	136
The *security_password_length* specifies a value less than 0 or greater than 10.	137
The *conversation_security_type* specifies an undefined value.	138
The *security_user_ID_length* specifies a value less than 0 or greater than 10.	139
The *conversation_type* specifies an undefined value.	140
The *conversation_type* specifies CM_MAPPED_CONVERSATION, and *fill* is set to CM_FILL_BUFFER.	141
The *conversation_type* specifies CM_MAPPED_CONVERSATION, and a prior call to Set_Log_Data is still in effect.	142

Table 65 (Page 3 of 11). CPI Communications-Defined Secondary Information

Condition Code	Description
The *deallocate_type* specifies CM_DEALLOCATE_FLUSH, *sync_level* is set to CM_SYNC_POINT, and *transaction_control* is set to CM_CHAINED_TRANSACTIONS.	143
The *deallocate_type* specifies CM_DEALLOCATE_FLUSH, *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, and *transaction_control* is set to CM_CHAINED_TRANSACTIONS.	144
The *deallocate_type* specifies CM_DEALLOCATE_CONFIRM, *sync_level* is set to CM_NONE, and the conversation is using an LU 6.2 CRM.	145
The *deallocate_type* specifies CM_DEALLOCATE_CONFIRM, *sync_level* is set to CM_SYNC_POINT, and the conversation is using an LU 6.2 CRM.	146
The *deallocate_type* specifies CM_DEALLOCATE_CONFIRM, *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, and the conversation is using an LU 6.2 CRM.	147
The *deallocate_type* specifies CM_DEALLOCATE_CONFIRM, *sync_level* is set to CM_SYNC_POINT, *transaction_control* is set to CM_CHAINED_TRANSACTIONS, and the conversation is using an OSI TP CRM.	148
The *deallocate_type* specifies CM_DEALLOCATE_CONFIRM, *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, *transaction_control* is set to CM_CHAINED_TRANSACTIONS, and the conversation is using an OSI TP CRM.	149
The *deallocate_type* specifies an undefined value.	150
The *error_direction* specifies CM_SEND_ERROR, and the conversation is using an OSI TP CRM.	151
The *error_direction* specifies an undefined value.	152
The *conversation_type* is set to CM_MAPPED_CONVERSATION.	153
The *fill* specifies an undefined value.	154
The *initialization_data_length* specifies a value less than 0 or greater than 10000.	155
The *conversation_type* is set to CM_MAPPED_CONVERSATION, and the conversation is using an LU 6.2 CRM.	156
The *log_data_length* specifies a value less than 0 or greater than 512.	157
The *map_name_length* specifies a value less than 0 or greater than 8.	158
The *partner_ID_type* specifies an undefined value.	159
The *partner_ID_type* specifies CM_PROGRAM_BINDING, and *partner_ID_length* specifies a value less than 0 or greater than 32767.	160
The *partner_ID_type* specifies CM_DISTINGUISHED_NAME or CM_PROGRAM_FUNCTION_ID, and *partner_ID_length* specifies a value less than 0 or greater than 1024.	161
The *partner_ID_scope* specifies an undefined value.	162

Table 65 (Page 4 of 11). CPI Communications-Defined Secondary Information

Condition Code	Description
The *directory_syntax* specifies an undefined value.	163
The *directory_encoding* specifies an undefined value.	164
The *partner_LU_name_length* specifies a value less than 1 or greater than 17.	165
The *prepare_data_permitted* specifies CM_PREPARE_DATA_PERMITTED, and the conversation is using an LU 6.2 CRM.	166
The *prepare_data_permitted* specifies an undefined value.	167
The *prepare_to_receive_type* specifies CM_PREP_TO_RECEIVE_CONFIRM, and *sync_level* set to CM_NONE.	168
The *prepare_to_receive_type* specifies CM_PREP_TO_RECEIVE_CONFIRM, and *sync_level* set to CM_SYNC_POINT_NO_CONFIRM.	169
The *prepare_to_receive_type* specifies an undefined value.	170
The *processing_mode* specifies an undefined value.	171
The program has chosen queue-level non-blocking for the conversation.	172
The *conversation_queue* specifies a value that is not defined for the *send_receive_mode* conversation characteristic.	173
The program has chosen conversation-level non-blocking for the conversation.	174
The *queue_processing_mode* specifies an undefined value.	175
The *receive_type* specifies an undefined value.	176
The *return_control* specifies an undefined value.	177
The *send_receive_mode* specifies CM_FULL_DUPLEX, and *sync_level* is set to CM_CONFIRM.	178
The *send_receive_mode* specifies CM_FULL_DUPLEX, and *sync_level* is set to CM_SYNC_POINT.	179
The *send_receive_mode* specifies CM_FULL_DUPLEX, and *send_type* is set to CM_SEND_AND_PREP_TO_RECEIVE.	180
The *send_receive_mode* specifies CM_FULL_DUPLEX, and the program has chosen conversation-level non-blocking for the conversation.	181
The *send_receive_mode* specifies an undefined value.	182
The *send_type* specifies CM_SEND_AND_CONFIRM, and *sync_level* is set to CM_NONE.	183
The *send_type* specifies CM_SEND_AND_CONFIRM, and *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM.	184
The *send_type* specifies CM_SEND_AND_CONFIRM, and *send_receive_mode* is set to CM_FULL_DUPLEX.	185
The *send_type* specifies CM_SEND_AND_PREP_TO_RECEIVE, and *send_receive_mode* is set to CM_FULL_DUPLEX.	186

Table 65 (Page 5 of 11). CPI Communications-Defined Secondary Information

Condition Code	Description
The *send_type* specifies an undefined value.	187
The *sync_level* specifies CM_NONE, *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, and the conversation is using an LU 6.2 CRM.	188
The *sync_level* specifies CM_NONE, *send_receive_mode* is set to CM_HALF_DUPLEX, and *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_CONFIRM.	189
The *sync_level* specifies CM_NONE, *send_receive_mode* is set to CM_HALF_DUPLEX, and *send_type* is set to CM_SEND_AND_CONFIRM.	190
The *sync_level* specifies CM_SYNC_POINT_NO_CONFIRM, *send_receive_mode* is set to CM_HALF_DUPLEX, and *send_type* is set to CM_SEND_AND_CONFIRM.	191
The *sync_level* specifies CM_CONFIRM, and *send_receive_mode* is set to CM_FULL_DUPLEX.	192
The *sync_level* specifies CM_SYNC_POINT, and *send_receive_mode* is set to CM_FULL_DUPLEX.	193
The *sync_level* specifies CM_SYNC_POINT, *deallocate_type* is set to CM_DEALLOCATE_FLUSH, and the conversation is using an LU 6.2 CRM.	194
The *sync_level* specifies CM_SYNC_POINT, *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, and the conversation is using an LU 6.2 CRM.	195
The *sync_level* specifies CM_SYNC_POINT_NO_CONFIRM, *deallocate_type* is set to CM_DEALLOCATE_FLUSH, and the conversation is using an LU 6.2 CRM.	196
The *sync_level* specifies CM_SYNC_POINT_NO_CONFIRM, *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, and the conversation is using an LU 6.2 CRM.	197
The *sync_level* specifies CM_SYNC_POINT_NO_CONFIRM, *send_receive_mode* is set to CM_HALF_DUPLEX, and the conversation is using an LU 6.2 CRM.	198
The *sync_level* specifies an undefined value.	199
The *transaction_control* specifies CM_UNCHAINED_TRANSACTIONS, and the conversation is using an LU 6.2 CRM.	200
The *transaction_control* specifies an undefined value.	201
The *TP_name_length* specifies a value less than 1 or greater than 64.	202
The *TP_name* specifies a name that is restricted in some way by node services.	203
The *TP_name* has incorrect internal syntax as defined by node services.	204
The *TP_name_length* specifies a value less than 1 or greater than 64.	205
The *OOID_list_count* specifies a value less than 1.	206

Table 65 (Page 6 of 11). CPI Communications-Defined Secondary Information

Condition Code	Description
The number of OOIDs in *OOID_list* is less than the value specified in *OOID_list_count*.	207
The *OOID_list* contains an unassigned OOID.	208
The *timeout* specifies a value less than 0.	209
The *send_length* calculated by the map routine, specifies a value less than < 1 or > 86.	210
The *conversation_type* is set to CM_BASIC_CONVERSATION.	211
The *transaction_control* is set to CM_CHAINED_TRANSACTION.	212 - 1000
Reserved for future conditions to be associated with CM_PROGRAM_PARAMETER_CHECK.	
Associated with CM_PROGRAM_STATE_CHECK:	
1001	
No incoming conversation exists.	1002
No name is associated with the program. A program associates a name with itself by issuing the Specify_Local_TP_Name call.	1003
The conversation is not in **Initialize-Incoming** state.	1004
The conversation is not in **Initialize** state.	1005
The conversation's context is in the **Backout-Required** condition.	1006
The conversation is not in **Send**, **Send-Pending**, or **Defer-Receive** state.	1007
The conversation is basic, and the program started but did not finish sending a logical record.	1008
The conversation is not in **Confirm**, **Confirm-Send**, or **Confirm-Deallocate** state.	1009
The conversation is not in **Confirm-Deallocate** state.	1010
The conversation is not in **Send** or **Send-Pending** state.	1011
The *deallocate_type* is set to CM_DEALLOCATE_FLUSH, and the conversation is currently included in a transaction.	1012
The *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, and the conversation is currently included in a transaction.	1013
The program has received a *status_received* value of CM_JOIN_TRANSACTION and must issue a tx_begin call to the X/Open TX interface to join the transaction.	1014
The conversation is not in **Send-Receive** or **Send-Only** state.	1015
The conversation is not in **Send-Receive** state.	1016
The conversation is not currently included in a transaction.	1017
The conversation is in **Initialize-Incoming** state.	1018
The conversation is in **Initialize** state.	1019

Table 65 (Page 7 of 11). CPI Communications-Defined Secondary Information

Condition Code	Description
The conversation's context is not in transaction. The program must issue a tx_begin call to the X/Open TX interface to start a transaction.	1020
The conversation is already included in the current transaction.	1021
The conversation is using an OSI TP CRM, *begin_transaction* is set to CM_BEGIN_EXPLICIT, and the conversation is not currently included in a transaction.	1022
The conversation is using an OSI TP CRM, and the program is not the root of the transaction and has not received a take-commit notification from its superior.	1023
A prior call to Deferred_Deallocate is still in effect for the conversation.	1024
The *receive_type* is set to CM_RECEIVE_AND_WAIT, and the conversation is not in **Send**, **Receive**, **Send-Pending**, or **Prepared** state.	1025
The *receive_type* is set to CM_RECEIVE_IMMEDIATE, and the conversation is not in **Receive** or **Prepared** state.	1026
The conversation is not in **Send-Receive**, **Receive-Only**, or **Prepared** state.	1027
The conversation is not in **Send**, **Receive**, **Send-Pending**, **Confirm**, **Confirm-Send**, **Confirm-Deallocate**, **Sync-Point**, **Sync-Point-Send**, **Sync-Point-Deallocate**, or **Prepared** state.	1028
For a conversation using an OSI TP CRM, the Request_To_Send call is not allowed from **Send** state.	1029
The conversation is not in **Send**, **Send-Pending**, **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** state.	1030
The program received a take-commit notification not ending in *_DATA_OK, and the conversation is in **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** state.	1031
The *send_type* is set to CM_SEND_AND_CONFIRM or CM_SEND_AND_PREP_TO_RECEIVE, and the conversation is in **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** state.	1032
The *send_type* is set to CM_SEND_AND_DEALLOCATE, *deallocate_type* is not set to CM_DEALLOCATE_ABEND, and the conversation is in **Sync-Point**, **Sync-Point-Send**, or **Sync-Point-Deallocate** state.	1033
The *send_type* is set to CM_SEND_AND_DEALLOCATE, *deallocate_type* is set to CM_DEALLOCATE_FLUSH, *sync_level* is set to CM_SYNC_POINT, and the conversation is included in a transaction.	1034
The *send_type* is set to CM_SEND_AND_DEALLOCATE, *deallocate_type* is set to CM_DEALLOCATE_FLUSH, *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, and the conversation is included in a transaction.	1035

Table 65 (Page 8 of 11). CPI Communications-Defined Secondary Information

Condition Code	Description
The *send_type* is set to CM_SEND_AND_DEALLOCATE, *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, *sync_level* is set to CM_SYNC_POINT, and the conversation is included in a transaction.	1036
The *send_type* is set to CM_SEND_AND_DEALLOCATE, *deallocate_type* is set to CM_DEALLOCATE_CONFIRM, *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, and the conversation is included in a transaction.	1037
The conversation is not in **Send-Receive**, **Send-Only**, **Sync-Point**, or **Sync-Point-Deallocate** state.	1038
The program received a take-commit notification not ending in *_DATA_OK, and the conversation is in **Sync-Point** or **Sync-Point-Deallocate** state.	1039
The *send_type* is set to CM_SEND_AND_DEALLOCATE, *deallocate_type* is not set to CM_DEALLOCATE_ABEND, and the conversation is in **Sync-Point** or **Sync-Point-Deallocate** state.	1040
The conversation is not in **Send-Receive**, **Send-Only**, or **Confirm-Deallocate** state.	1041
The *conversation_security_type* is not set to CM_SECURITY_PROGRAM or CM_SECURITY_PROGRAM_STRONG.	1042
The conversation is not in **Initialize** or **Initialize-Incoming** state.	1043
The conversation is not in **Initialize** or **Receive** state.	1044
The conversation is not in **Initialize** or **Send-Receive** state.	1045
The *conversation_queue* specifies CM_INITIALIZATION_QUEUE, and the conversation is not in **Initialize** or **Initialize-Incoming** state.	1046
The *conversation_queue* specifies a value other than CM_INITIALIZATION_QUEUE, and the conversation is in **Initialize-Incoming** state.	1047
The conversation is not in **Send**, **Receive**, **Send-Pending**, **Defer-Receive**, or **Defer-Deallocate** state.	1048
There is no outstanding operation associated with any of the OOIDs specified in *OOID_list* or by use of a defined value of *OOID_list_count* has completed.	1049
There were no conversation-level outstanding operations for the program.	1050
The *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, *transaction_control* is set to CM_CHAINED_TRANSACTIONS, and the conversation's context is not in transaction.	1051
The program has issued a successful Accept_Conversation or Accept_Incoming call on a conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and using an OSI TP CRM, and the program has not issued a Receive call on this conversation.	1052 - 2000

Table 65 (Page 9 of 11). CPI Communications-Defined Secondary Information

Condition Code	Description
Reserved for future conditions to be associated with CM_PROGRAM_STATE_CHECK.	
Associated with CM_PARAMETER_ERROR:	
2001	
The *mode_name* characteristic (set from side information or by Set_Mode_Name) specifies a mode name that is not recognized by the LU as being valid.	2002
The *mode_name* characteristic (set from side information or by Set_Mode_Name) specifies a mode name that the local program does not have the authority to specify. For example, SNASVCMG requires special authority with LU 6.2.	2003
The *TP_name* characteristic (set from side information or by Set_TP_Name) specifies a transaction program name that the local program does not have the appropriate authority to allocate a conversation to. For example, SNA service programs require special authority with LU 6.2.	2004
The *TP_name* characteristic (set from side information or by Set_TP_Name) specifies an SNA service transaction program and *conversation_type* is set to CM_MAPPED_CONVERSATION.	2005
The *partner_LU_name* characteristic (set from side information or by Set_Partner_LU_Name) specifies a partner LU name that is not recognized as being valid.	2006
The *AP_title* characteristic (set from side information or using Set_AP_Title call or the *AE_qualifier* characteristic (set from side information or using Set_AE_Qualifier call), or the *application_context_name* characteristic (set from side information or using the Set_Application_Context_Name call) specifies an AP title or an AE qualifier or an application context name that is not recognized as being valid.	2007
The *conversation_security_type* characteristic is set to CM_SECURITY_PROGRAM or CM_SECURITY_PROGRAM_STRONG, and the *security_password* characteristic or the *security_user_ID* characteristic (set from side information or by Set calls) or both, are null.	2008
The *conversation_security_type* characteristic is set to CM_SECURITY_DISTRIBUTED or CM_SECURITY_MUTUAL, and the partner principal name (set from the program binding) is null or is not recognized by the CRM as being valid.	2009
A *partner_ID* characteristic was provided that caused a search of the distributed directory, but no program binding was retrieved.	2010
The program binding for the conversation, either specified directly on the Set_Partner_ID call or obtained from the distributed directory, was invalid.	2011 - 2500

Table 65 (Page 10 of 11). CPI Communications-Defined Secondary Information

Condition Code	Description
Reserved for future conditions to be associated with CM_PARAMETER_ERROR.	
Associated with CM_SECURITY_NOT_SUPPORTED:	
2501	
The *conversation_security_type* does not provide for the user name type indicated by the required_user_name_type field in the program binding.	2502 - 3000
Reserved for future conditions to be associated with CM_SECURITY_NOT_SUPPORTED.	
Associated with CM_DEALLOCATED_ABEND (full-duplex conversations using an OSI TP CRM only):	
3001	
There was a collision between a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_CONFIRM issued by the local program and an Include_Partner_In_Transaction call issued by the partner program. No log data is available.	3002
There was a collision between a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_CONFIRM issued by the local program and a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_CONFIRM call issued by the partner program. No log data is available.	3003
CPI Communications deallocated the incoming conversation because an implicit call of tx_set_transaction_control failed with TX return code TX_PROTOCOL_ERROR.	3004
CPI Communications deallocated the incoming conversation because an implicit call of tx_set_transaction_control failed with TX return code TX_FAIL.	3005
CPI Communications deallocated the conversation because an implicit call of tx_begin failed with TX return code TX_OUTSIDE.	3006
CPI Communications deallocated the conversation because an implicit call of tx_begin failed with TX return code TX_PROTOCOL_ERROR.	3007
CPI Communications deallocated the conversation because an implicit call of tx_begin failed with TX return code TX_ERROR.	3008
CPI Communications deallocated the conversation because an implicit call of tx_begin failed with TX return code TX_FAIL.	3009 - 3500
Reserved for future conditions to be associated with CM_DEALLOCATED_ABEND.	
Associated with CM_DEALLOCATED_ABEND_BO (full-duplex conversations using an OSI TP CRM only):	
3501	

Table 65 (Page 11 of 11). CPI Communications-Defined Secondary Information

Condition Code	**Description**
There was a collision between a Include_Partner_In_Transaction call issued by the local program and a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_CONFIRM issued by the partner program. No log data is available.	3502 - 4000
Reserved for future conditions to be associated with CM_DEALLOCATED_ABEND_BO.	

CRM-Specific Secondary Information

When the underlying CRM discovers an abnormal condition, the condition, identified by a CRM-specific message, is then mapped to a CPI Communications return code and returned to the program. The CRM-specific message is the SNA sense data information for the CRM type of LU 6.2 and OSI diagnostic information for the CRM type of OSI TP.

Table 66. Examples of LU 6.2 CRM-Specific Secondary Information

Associated with CM_CONVERSATION_TYPE_MISMATCH:	4001;;1008 6034 The FMH-5 Attach command specifies a conversation type that the receiver does not support for the specified transaction program. This sense data is sent only in FMH-7.
Associated with CM_TPN_NOT_RECOGNIZED:	4001;;1008 6021 Transaction Program Name Not Recognized: The FMH-5 Attach command specifies a transaction program name that the receiver does not recognize. This sense data is sent only in FMH-7.
Associated with CM_SYNC_LVL_NOT_SUPPORTED_SYS:	4001;;1008 6040 Invalid Attach Parameter: A parameter in the FMH-5 Attach command conflicts with the statements of LU capability previously provided in the BIND negotiation.

Note: See Chapter 10 of *System Network Architecture Formats* (IBM document number GA27-3136) for complete information about sense data.

Table 67. Examples of OSI TP CRM-Specific Secondary Information

Associated with CM_RESOURCE_FAILURE_NO_RETRY, CM_RESOURCE_FAIL_NO_RETRY_BO:	4002;;recipient-unknown. 4002;;no-reason-given. 4002;;permanent-failure. 4002;;protocol-error.
Associated with CM_TPN_NOT_RECOGNIZED:	4002;;recipient-tpsu-title-unknown. 4002;;recipient-tpsu-title-required.
Associated with CM_SYNC_LEVEL_NOT_SUPPORTED_SYS:	4002;;functional-unit-not-supported. 4002;;functional-unit-combination-not-supported.

Implementation-Related Information

An implementation may return CM_PRODUCT_SPECIFIC_ERROR to the program for any errors that are specific to the implementation or to the system that supports the implementation. In this case, secondary information is the error message defined by the implementation or system.

Table 68. Examples of Implementation-Related Secondary Information

Example 1:
4003;;0001 STACK_TOO_SMALL;;A minimum stack size of 3500 bytes is required by CPI-C when a call is issued. CPI-C runs on the stack of the program that calls it. When the call was issued, CPI-C found the stack size to be less than the minimum size.;;Programmer Response: Increase the stack size specified in the .DEF file used in linking. If your program calls CPI-C from a thread it has created, be sure the stack size on the NewThreadStack parameter of your DosCreateThread function call is large enough.
Example 2:
4003;;0002 CANNOT_ALLOCATE_SHARED_SEGMENT;;Communications Manager could not allocate the shared segment named \SHAREMEM\ACSLGMEM.;;Programmer Response: A necessary shared segment is not currently available. There is no corrective action that your program can take. This problem will recur until the Communications Manager is stopped and restarted. Operator Response: Communications Manager must be restarted to correct the problem.

Appendix C. State Tables

The CPI Communications state tables show when and where different CPI Communications calls can be issued. For example, a program must issue an Initialize_Conversation call before issuing an Allocate call, and it cannot issue a Send_Data call before the conversation is allocated.

As described in "Program Flow—States and Transitions" on page 52, CPI Communications uses the concepts of states and state transitions to simplify explanations of the restrictions that are placed on the calls. A number of states are defined for CPI Communications and, for any given call, a number of transitions are allowed.

- Table 69 on page 704 describes the state transitions that are allowed for the CPI Communications calls on half-duplex conversations.
- Table 72 on page 718 describes the state transitions that are allowed for CPI Communications calls on full-duplex conversations.
- Table 70 on page 710 shows the effects of SAA resource recovery Commit and Backout calls on CPI Communications conversation states for half-duplex conversations.
- Table 71 on page 711 shows the effects of X/Open TX resource recovery calls on CPI Communications conversation states for half-duplex conversations.
- Table 73 on page 723 shows the effects of SAA resource recovery calls on CPI Communications conversation states for full-duplex conversations.
- Table 74 on page 724 shows the effects of X/Open TX resource recovery calls on CPI Communications conversation states for full-duplex conversations.

How to Use the State Tables

Each CPI Communications call[8] is represented in the table by a group of input rows. The possible conversation states are shown across the top of the table. The states correspond to the columns of the matrix. The intersection of input (row) and state (column) represents the validity of a CPI Communications call in that particular state and, for valid calls, what state transition (if any) occurs.

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

8 Only the calls that affect conversation states are included in the State table.

© Copyright IBM Corp. 1996, 1998

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

For calls that are processed in non-blocking processing mode, the following special considerations apply:

- When a call gets the CM_OPERATION_INCOMPLETE return code, the operation remains in progress as an oustanding operation on the conversation (when conversation-level non-blocking is used) or on the queue with which the call is associated (when queue-level non-blocking is used). Any other calls (except Cancel_Conversation) on that conversation or queue get a return code of CM_OPERATION_NOT_ACCEPTED, and no conversation state transition occurs.
- The CM_OPERATION_NOT_ACCEPTED return code is not included in the state table.

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

- A state transition symbol ending with a caret (for example, 1^ or –^) means that the conversation's context may be in the **Backout-Required** condition following the call. (Note that the state change for the conversation is indicated by the **first** character of these symbols.)
- When a context is in the **Backout-Required** condition, its protected conversations are restricted from issuing certain CPI Communications calls. These calls are designated in the table with the symbol ↓′. Where this symbol appears, the call is valid in this state unless the conversation is protected and its context is in the **Backout-Required** condition. If the call is invalid, a *return_code* of CM_PROGRAM_STATE_CHECK is returned and no conversation state transition occurs.

Example

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

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

- The conversation goes from **Send** or **Send-Pending** state to **Reset** state (state 1) when a return code abbreviated as “ok,” “da,” or “rf” is returned. See “Return Code Values [ ]” on page 700 to find out what these abbreviations mean.

- The conversation goes from **Send** state to **Reset** state when a return code abbreviated as “ae” is returned. A return code abbreviated as “ae” will never be returned when this call is issued from **Send-Pending** state.
- The conversation goes from **Send** or **Send-Pending** state to **Receive** state (state 4) when a return code abbreviated as “ep” is returned.
- There is no state transition when a return code of CM_PROGRAM_PARAMETER_CHECK (“pc”) or CM_OPERATION_INCOMPLETE (“oi”) is returned.
- There is no state transition for a conversation in **Send** state when a return code of CM_PROGRAM_STATE_CHECK (“sc”) is returned. This return code will never be returned when this call is issued from **Send-Pending** state.

Explanation of Half-Duplex State Table Abbreviations

Abbreviations are used in the state table to indicate the different permutations of calls and characteristics. There are four categories of abbreviations:

- **Conversation characteristic** abbreviations are enclosed by parentheses— (. . .)
- **Conversation queue** abbreviations are enclosed by parentheses— (. . .)
- **return_code** abbreviations are enclosed by brackets — [. . .]
- **data_received and status_received** abbreviations are enclosed by braces and separated by a comma— { . . . , . . . }. The abbreviation before the comma represents the *data_received* value, and the abbreviation after the comma represents the value of *status_received*.

The next sections show the abbreviations used in each category.

Conversation Characteristics ( )

The following abbreviations are used for conversation characteristics:

Abbreviation	Meaning
A	*deallocate_type* is set to CM_DEALLOCATE_ABEND
B	*send_type* is set to CM_BUFFER_DATA

Abbreviation	Meaning
C	For a Deallocate call, C means one of the following: • *deallocate_type* is set to CM_DEALLOCATE_CONFIRM • *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is set to CM_CONFIRM • *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is set to CM_SYNC_POINT, but the conversation is not currently included in a transaction For a Prepare_To_Receive call, C means one of the following: • *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_CONFIRM • *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* is set to CM_CONFIRM • *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* is set to CM_SYNC_POINT, but the conversation is not currently included in a transaction For a Send_Data call, C means the following: • *send_type* is set to CM_SEND_AND_CONFIRM
D(x)	*send_type* is set to CM_SEND_AND_DEALLOCATE. *x* represents the *deallocate_type* and can be A, C, F, or S. Refer to the appropriate entries in this table for a description of these values.
F	For a Deallocate call, F means one of the following: • *deallocate_type* is set to CM_DEALLOCATE_FLUSH • *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and either *sync_level* is set to CM_NONE or the conversation is in **Initialize_Incoming** state • *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, but the conversation is not currently included in a transaction For a Prepare_To_Receive call, F means one of the following: • *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_FLUSH • *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* is set to CM_NONE • *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, but the conversation is not currently included in a transaction For a Send_Data call, F means the following: • *send_type* is set to CM_SEND_AND_FLUSH
I	*receive_type* is set to CM_RECEIVE_IMMEDIATE
P(x)	*send_type* is set to CM_SEND_AND_PREP_TO_RECEIVE. *x* represents the *prepare_to_receive_type* and can be C, F, or S. Refer to the appropriate entries in this table for a description of these values.

Abbreviation	Meaning
S	For a Deallocate call, S means the following: • *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and the conversation is currently included in a transaction For a Prepare_To_Receive call, S means the following: • *prepare_to_receive_type* is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, and the conversation is currently included in a transaction
W	*receive_type* is set to CM_RECEIVE_AND_WAIT.

Conversation Queues ()

The following abbreviations are used for conversation queues:

Abbreviation	Meaning
N	*conversation_queue* is set to CM_INITIALIZATION_QUEUE
Q	*conversation_queue* is set to one of the following: • CM_SEND_RECEIVE_QUEUE • CM_EXPEDITED_SEND_QUEUE • CM_EXPEDITED_RECEIVE_QUEUE

Return Code Values [ ]

The following abbreviations are used for return codes:

Abbreviation	Meaning
ae	For an Allocate call, ae means one of the following: • CM_ALLOCATE_FAILURE_NO_RETRY • CM_ALLOCATE_FAILURE_RETRY • CM_CONVERSATION_TYPE_MISMATCH • CM_PIP_NOT_SPECIFIED_CORRECTLY • CM_RESOURCE_FAILURE_NO_RETRY • CM_RESOURCE_FAILURE_RETRY • CM_RETRY_LIMIT_EXCEEDED • CM_SECURITY_MUTUAL_FAILED • CM_SECURITY_NOT_SUPPORTED • CM_SECURITY_NOT_VALID • CM_SEND_RCV_MODE_NOT_SUPPORTED • CM_SYNC_LVL_NOT_SUPPORTED_PGM • CM_SYNC_LVL_NOT_SUPPORTED_SYS • CM_TP_NOT_AVAILABLE_NO_RETRY • CM_TP_NOT_AVAILABLE_RETRY • CM_TPN_NOT_RECOGNIZED For any other call, ae means one of the following: • CM_CONVERSATION_TYPE_MISMATCH • CM_PIP_NOT_SPECIFIED_CORRECTLY • CM_SECURITY_NOT_VALID • CM_SEND_RCV_MODE_NOT_SUPPORTED • CM_SYNC_LVL_NOT_SUPPORTED_PGM • CM_SYNC_LVL_NOT_SUPPORTED_SYS • CM_TP_NOT_AVAILABLE_NO_RETRY • CM_TP_NOT_AVAILABLE_RETRY • CM_TPN_NOT_RECOGNIZED
bo	CM_TAKE_BACKOUT. This return code is returned only for conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.
bs	CM_BUFFER_TOO_SMALL
da	da means one of the following: • CM_DEALLOCATED_ABEND • CM_DEALLOCATED_ABEND_SVC • CM_DEALLOCATED_ABEND_TIMER
db	db is returned only for conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and means one of the following: • CM_DEALLOCATED_ABEND_BO • CM_DEALLOCATED_ABEND_SVC_BO • CM_DEALLOCATED_ABEND_TIMER_BO
dn	CM_DEALLOCATED_NORMAL
dnb	CM_DEALLOCATED_NORMAL_BO. This return code is returned only for conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.
ed	ed means one of the following: • CM_EXP_DATA_NOT_SUPPORTED • CM_CONVERSATION_ENDING

Abbreviation	Meaning
en	en means one of the following: • CM_PROGRAM_ERROR_NO_TRUNC • CM_SVC_ERROR_NO_TRUNC
ep	ep means one of the following: • CM_PROGRAM_ERROR_PURGING • CM_SVC_ERROR_PURGING
et	et means one of the following: • CM_PROGRAM_ERROR_TRUNC • CM_SVC_ERROR_TRUNC
mp	mp means one of the following: • CM_UNKNOWN_MAP_NAME_REQUESTED • CM_UNKNOWN_MAP_NAME_RECEIVED • CM_MAP_ROUTINE_ERROR
ns	CM_NO_SECONDARY_INFORMATION
oi	CM_OPERATION_INCOMPLETE
ok	CM_OK
pb	CM_INCLUDE_PARTNER_REJECT_BO
pc	CM_PROGRAM_PARAMETER_CHECK. This return code means an error was found in one or more parameters. For calls illegally issued in **Reset** state, pc is returned because the *conversation_ID* is undefined in that state.
pe	CM_PARAMETER_ERROR
pn	CM_PARM_VALUE_NOT_SUPPORTED
rb	rb means one of the following: • CM_RESOURCE_FAIL_NO_RETRY_BO • CM_RESOURCE_FAILURE_RETRY_BO
rf	rf means one of the following: • CM_RESOURCE_FAILURE_NO_RETRY • CM_RESOURCE_FAILURE_RETRY
sc	CM_PROGRAM_STATE_CHECK
se	CM_SYSTEM_EVENT
un	CM_UNSUCCESSFUL

Notes:

1. The return code CM_PRODUCT_SPECIFIC_ERROR is not included in the state table because the state transitions caused by this return code are product-specific.

2. The CM_OPERATION_NOT_ACCEPTED return code is not included in the state table. If conversation-level non-blocking is being used on a conversation, a program receives CM_OPERATION_NOT_ACCEPTED when it issues any call (except Cancel_Conversation) on the conversation while a previous operation is still in progress, regardless of the state. If conversation-level non-blocking is not being used on a conversation, a program receives CM_OPERATION_NOT_ACCEPTED when it issues any call associated with a

queue that has a previous operation still in progress, regardless of the state. No conversation state transition occurs.

3. The CM_CALL_NOT_SUPPORTED return code is not included in the state table. It is returned when the local system provides an entry point for the call but does not support the function requested by the call, regardless of the state. No state transition occurs.

data_received and status_received { , }

The following abbreviations are used for the *data_received* values:

Abbreviation	Meaning
dr	Means one of the following: • CM_DATA_RECEIVED • CM_COMPLETE_DATA_RECEIVED • CM_INCOMPLETE_DATA_RECEIVED
nd	CM_NO_DATA_RECEIVED
*	Means one of the following: • CM_DATA_RECEIVED • CM_COMPLETE_DATA_RECEIVED • CM_NO_DATA_RECEIVED

The following abbreviations are used for the *status_received* values:

Abbreviation	Meaning
cd	CM_CONFIRM_DEALLOC_RECEIVED
co	CM_CONFIRM_RECEIVED
cs	CM_CONFIRM_SEND_RECEIVED
jt	CM_JOIN_TRANSACTION
no	CM_NO_STATUS_RECEIVED
po	CM_PREPARE_OK
se	CM_SEND_RECEIVED
tc	CM_TAKE_COMMIT or CM_TAKE_COMMIT_DATA_OK. These values are returned only for conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.
td	CM_TAKE_COMMIT_DEALLOCATE or CM_TAKE_COMMIT_DEALLOC_DATA_OK. These values are returned only for conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.
ts	CM_TAKE_COMMIT_SEND or CM_TAKE_COMMIT_SEND_DATA_OK. These values are returned only for conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.

Table Symbols for the Half-Duplex State Table

The following symbols are used in the state table to indicate the condition that results when a call is issued from a certain state:

Symbol	Meaning
/	Cannot occur. CPI Communications either will not allow this input or will never return the indicated return codes for this input in this state.
–	Remain in current state
1-18	Number of next state
↓	It is valid to make this call from this state. See the table entries immediately below this symbol to determine the state transition resulting from the call.
↓′	For a conversation not using *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, or not currently included in a transaction, this is equivalent to ↓. If the conversation has *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and the conversation is currently included in a transaction, however, ↓′ means it is valid to make this call from this state *unless the conversation's context is in the* **Backout-Required** *condition*. In that case, the call is invalid and CM_PROGRAM_STATE_CHECK is returned. For valid calls, see the table entries immediately below this symbol to determine the state transition resulting from the call.
^	For a conversation not using *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM or not currently included in a transaction, this symbol should be ignored. For a conversation using *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and currently included in a transaction, when this symbol follows a state number or a – (for example, 1^ or –^), it means the conversation's context may be in the **Backout-Required** condition following the call.
#	A conversation with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM and currently included in a transaction goes to the state it was in at the completion of the most recent synchronization point. If there was no prior synchronization event, the side of the conversation that was initialized with an Allocate call goes to **Send** state, and the side of the conversation that accepted the conversation goes to **Receive** state. On CICS/ESA systems, when the program is using CPI Communications to communicate with releases of CICS earlier than CICS/ESA Version 3.2, the program's conversation state after a backout is one of the following: • If the program initiated the backout, its side of the conversation is placed in **Send** state. • If the program did not initiate the backout, its side of the conversation is placed in **Receive** state. • If the program's side of the conversation was in **Defer-Deallocate** state when the backout occurred, the conversation is placed in **Reset** state.
%	Wait_For_Completion and Wait_For_Conversation can only be issued when one or more calls have received a *return_code* of CM_OPERATION_INCOMPLETE. When Wait_For_Completion or Wait_For_Conversation completes with a *return_code* of CM_OK, it indicates one or more conversations on which an operation has completed. Each of those conversations then moves to the appropriate state as determined by the return code for the operation that is now completed and by the other factors that determine state transitions.
?	For programs using the X/Open TX interface with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM, the state of the conversation with respect to the transaction is unknown.

Table 69 (Page 1 of 6). States and Transitions for CPI Communications Calls on Half-Duplex Conversations

Inputs	States 1-8 and 14 are used by all conversations								Used only by conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM						
	Reset 1	Ini- tialize 2	Send 3	Re- ceive 4	Send- Pend- ing 5	Con- firm 6	Con- firm- Send 7	Con- firm- Deal- locate 8	Defer- Re- ceive 9	Defer- Deal- locate 10	Sync- Point 11	Sync- Point Send 12	Sync- Point Deal- locate 13	Pre- pared 18	Ini- tialize- In- coming 14
Accept_Conversation	↓	/	/	/	/	/	/	/	/	/	/	/	/	/	/
[ok]	4														
[da,sc]	–														
Accept_Incoming	[pc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	↓
[ok]															4
[da]															1
[oi,pc,sc]															–
Allocate	[pc]	↓'	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]		3													
[ae]		1													
[oi,pc,pe,sc,un]		–													
Cancel_Conversation	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok]		1^	1^	1^	1^	1^	1^	1^	1^	1^	1^	1^	1^	1^	1^
[pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	–
Confirm	[pc]	[sc]	↓'	[sc]	↓'	[sc]	[sc]	[sc]	↓'	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]			–		3				4						
[ae]			1		/				1						
[bo]			–^		3^				4^						
[da,rf]			1		1				1						
[db,pb,rb]			1^		1^				1^						
[ep]			4		4				4						
[oi,pc]			–		–				–						
[sc]			–		/				/						
Confirmed	[pc]	[sc]	[sc]	[sc]	[sc]	↓'	↓'	↓'	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]						4	3	1							
[oi,pc]						–	–	–							
Deallocate(A)	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	/
[ok]		1	1^	1^	1^	1^	1^	1^	1^	1^	1^	1^	1^	1^	
[oi,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Deallocate(C)	[pc]	[sc]	↓	[sc]	↓	[sc]	[sc]	[sc]	/	/	/	/	/	/	/
[ok,da,rf]			1		1										
[ae]			1		/										
[ep]			4		4										
[oi,pc]			–		–										
[sc]			–		/										
Deallocate(F)	[pc]	[sc]	↓	[sc]	↓	[sc]	[sc]	[sc]	/	/	/	/	/	/	↓
[ok]			1		1										1
[oi,pc]			–		–										–
[sc]			–		/										/
Deallocate(S)	[pc]	[sc]	↓'	[sc]	↓'	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	/
[ok]			10		10										
[oi,pc]			–		–										
[sc]			–		/										
Deferred_Deallocate[9]	[pc]	[sc]	↓'	[sc]	↓'	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]			–[10]		–[10]										
[ae]			1		/										
[bo]			–^		3^										
[db,rb]			1^		1^										
[ep]			4		4										
[oi,pc]			–		–										
[pb]			1^		/										
Extract_AE_Qualifier	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Extract_AP_Title	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Extract_Appl_Ctx_Name	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Extract_Conv_Context	[pc]	[sc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,sc]			–	–	–	–	–	–	–	–	–	–	–	–	
Extract_Conv_State	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	–
[bo]		/	–	–	–	–	–	/	–	–	–	–	–	–	/
Extract_Conv_Type	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	

Table 69 (Page 2 of 6). States and Transitions for CPI Communications Calls on Half-Duplex Conversations

Extract_Init_Data	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,bs,pc,bs]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Extract_Mapped_ Initialization_Data	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc,mp,bs]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Extract_Mode_Name	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Extract_Part_LU_Name	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Extract_Partner_ID	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,bs,pc]	–	–	–	–	–	–	–	–	–	–	–	–	–	–	
Extr_Sec_User_ID	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Extract_Secondary_Info	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,ns,pc]	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Extr_Send_Rcv_Mode	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Extract_Sync_Level	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Extr_Transaction_Control	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Extract_TP_Name	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Flush	[pc]	[sc]	↓′	[sc]	↓′	[sc]	[sc]	[sc]	↓′	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]			–		3				4						
[oi,pc]			–		–				–						
Include_Ptr_In_Trans[9]	[pc]	[sc]	↓′	[sc]	↓′	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]			–		3										
[ae]			1		/										
[da,rf]			1		1										
[ep]			4		4										
[oi,pc,sc]			–		–										
Init_Conversation[11]	↓	/	/	/	/	/	/	/	/	/	/	/	/	/	/
[ok]	2														
[pc]	–														
Init_For_Incoming	↓	/	/	/	/	/	/	/	/	/	/	/	/	/	/
[ok]	14														
Prepare	[pc]	[sc]	↓′	[sc]	↓′	[sc]	[sc]	[sc]	↓′	↓′	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]			18		18				18	18					
[ae]			1		/				1	1					
[bo]			–^		–^				–^	–^					
[db,pb,rb]			1^		1^				1^	1^					
[ep]			4		4				4	4					
[oi,pc,sc]			–		–				–	–					
Prepare_To_Receive(C)	[pc]	[sc]	↓′	[sc]	↓′	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,ep]			4		4										
[ae]			1		/										
[bo]			4^		4^										
[da,rf]			1		1										
[db,pb,rb]			1^		1^										
[oi,pc]			–		–										
[sc]			–		/										
Prepare_To_Receive(F)	[pc]	[sc]	↓′	[sc]	↓′	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]			4		4										
[oi,pc]			–		–										
[sc]			–		/										
Prepare_To_Receive(S)	[pc]	[sc]	↓′	[sc]	↓′	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]			9		9										
[oi,pc]			–		–										
[sc]			–		/										

Table 69 (Page 3 of 6). States and Transitions for CPI Communications Calls on Half-Duplex Conversations

Receive(I)	[pc]	[sc]	[sc]	↓′	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	↓′	[sc]
[ok] {dr,no}				–										–	
[ok] {dr,se}				5										/	
[ok] {nd,se}				3										/	
[ok] {*,cd}				8										/	
[ok] {*,co}				6										/	
[ok] {*,cs}				7										/	
[ok] {*,jt}				–										/	
[ok] {*,po}				/										–	
[ok] {*,tc}				11										/	
[ok] {*,td}				13										/	
[ok] {*,ts}				12										/	
[ae]				1										1	
[bo]				–^										–^	
[da,dn,rf]				1										/	
[db,pb,rb]				1^										1^	
[en,ep]				–										4	
[et]				–										/	
[pc,sc,un]				–										–	
Receive(W)	[pc]	[sc]	↓′	↓′	↓′	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	↓′	[sc]
[ok] {dr,no}			4	–	4									–	
[ok] {dr,se}			5	5	–									/	
[ok] {nd,se}			–	3	3									/	
[ok] {*,cd}			8	8	8									/	
[ok] {*,co}			6	6	6									/	
[ok] {*,cs}			7	7	7									/	
[ok] {*,jt}			–	–	/									/	
[ok] {*,po}			/	/	/									–	
[ok] {*,tc}			11	11	11									/	
[ok] {*,td}			13	13	13									/	
[ok] {*,ts}			12	12	12									/	
[ae]			1	1	/									1	
[bo]			4^	–^	4^									–^	
[da,dn,rf]			1	1	1									/	
[db,pb,rb]			1^	1^	1^									1^	
[en,ep]			4	–	4									4	
[et]			/	–	/									/	
[oi,pc]			–	–	–									–	
[sc]			–	–	/									–	
Rcv_Exp_Data	[pc]	[sc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,bs,ed,oi,pc]			–	–	–	–	–	–	–	–	–	–	–	–	
Receive_Mapped_Data(I)	[pc]	[sc]	[sc]	↓′	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	↓′	[sc]
[ok] {dr,no}				–										–	
[ok] {dr,se}				5										/	
[ok] {nd,se}				3										/	
[ok] {*,cd}				8										/	
[ok] {*,co}				6										/	
[ok] {*,cs}				7										/	
[ok] {*,jt}				–										/	
[ok] {*,po}				/										–	
[ok] {*,tc}				11										/	
[ok] {*,td}				13										/	
[ok] {*,ts}				12										/	
[ae]				1										1	
[bo]				–^										–^	
[da,dn,rf]				1										/	
[db,pb,rb]				1^										1^	
[en,ep]				–										4	
[et]				–										/	
[pc,sc,un]				–										–	

Table 69 (Page 4 of 6). States and Transitions for CPI Communications Calls on Half-Duplex Conversations

Receive_Mapped_Data(W)	[pc]	[sc]	↓'	↓'	↓'	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	↓'	[sc]
[ok] {dr,no}			4	–	4									–	
[ok] {dr,se}			5	5	–									/	
[ok] {nd,se}			–	3	3									/	
[ok] {*,cd}			8	8	8									/	
[ok] {*,co}			6	6	6									/	
[ok] {*,cs}			7	7	7									/	
[ok] {*,jt}			–	–	/									/	
[ok] {*,po}			/	/	/									–	
[ok] {*,tc}			11	11	11									/	
[ok] {*,td}			13	13	13									/	
[ok] {*,ts}			12	12	12									/	
[ae]			1	1	/									1	
[bo]			4^	–^	4^									–^	
[da,dn,rf]			1	1	1									/	
[db,pb,rb]			1^	1^	1^									1^	
[en,ep]			4	–	4									4	
[et]			/	–	/									/	
[oi,pc]			–	–	–									–	
[sc]			–	–	/									–	
Request_To_Send[12]	[pc]	[sc]	↓'	↓'	↓'	↓'	↓'	↓'	[sc]	[sc]	↓'	↓'	↓'	↓'	[sc]
[oi,ok,pc]			–	–	–	–	–	–			–	–	–	–	
[sc]			/	–	/	/	/	/			/	/	/	/	
Send_Data	[pc]	[sc]	↓'	[sc]	↓'	[sc]	[sc]	[sc]	[sc]	[sc]	↓'	↓'	↓'	[sc]	[sc]
(B) [ok]			–		3						–	–	–		
(C) [ok]			–		3						/	/	/		
(D(A)) [ok]			1^		1^						1^	1^	1^		
(D(C)) [ok]			1		1						/	/	/		
(D(F)) [ok]			1		1						/	/	/		
(D(S)) [ok]			10		10						/	/	/		
(F) [ok]			–		3						–	–	–		
(P(C)) [ok]			4		4						/	/	/		
(P(F)) [ok]			4		4						/	/	/		
(P(S)) [ok]			9		9						/	/	/		
[ae]			1		/						/	/	/		
[bo]			–^		3^						–^	–^	–^		
[da,rf]			1		1						/	/	/		
[db,rb]			1^		1^						1^	1^	1^		
[ep]			4		4						/	/	/		
[oi,pc]			–		–						–	–	–		
[pb]			1^		/						/	/	/		
[sc]			–		/						–	–	–		
Send_Error	[pc]	[sc]	↓'	↓'	↓'	↓'	↓'	↓'	[sc]	[sc]	↓'	↓'	↓'	[sc]	[sc]
[ok]			–	3	3	3	3	3			3	3	3		
[ae,da]			1	1	/	/	/	/			/	/	/		
[bo]			–^	–^	3^	/	/	/			–^	–^	–^		
[db]			1^	1^	/	/	/	/			/	/	/		
[dn]			/	1	/	/	/	/			/	/	/		
[dnb]			/	1^	/	/	/	/			/	/	/		
[ep]			4	–	/	/	/	/			/	/	/		
[oi,pc]			–	–	–	–	–	–			–	–	–		
[pb]			1^	1^	/	/	/	/			/	/	/		
[rb]			1^	1^	1^	1^	1^	1^			1^	1^	1^		
[rf]			1	1	1	1	1	1			1	1	1		
[sc]			–	–	/	/	/	/			–	–	–		
Send_Exp_Data	[pc]	[sc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,ed,oi,pc]			–	–	–	–	–	–	–	–	–	–	–	–	
Send_Mapped_Data	[pc]	[sc]	↓'	[sc]	↓'	[sc]	[sc]	[sc]	[sc]	[sc]	↓'	↓'	↓'	[sc]	[sc]

Table 69 (Page 5 of 6). States and Transitions for CPI Communications Calls on Half-Duplex Conversations															
[mp]															
(B) [ok]			–		3						–	–	–		
(C) [ok]			–		3						/	/	/		
(D(A)) [ok]			1^		1^						1^	1^	1^		
(D(C)) [ok]			1		1						/	/	/		
(D(F)) [ok]			1		1						/	/	/		
(D(S)) [ok]			10		10						/	/	/		
(F) [ok]			–		3						–	–	–		
(P(C)) [ok]			4		4						/	/	/		
(P(F)) [ok]			4		4						/	/	/		
(P(S)) [ok]			9		9						/	/	/		
[ae]			1		/						/	/	/		
[bo]			–^		3^						–^	–^	–^		
[da,rf]			1		1						/	/	/		
[db,rb]			1^		1^						1^	1^	1^		
[ep]			4		4						/	/	/		
[oi,pc]			–		–						–	–	–		
[pb]			1^		/						/	/	/		
[sc]			–		/						–	–	–		
Set_AE_Qualifier	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–													
Set_Allocate_Confirm	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–													
Set_AP_Title	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–													
Set_Appl_Context_Name	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–													
Set_Begin_Transaction	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Set_Confirmation_Urgency	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	–
Set_Conv_Sec_PW	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–													
Set_Conv_Sec_Type	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc,pn]		–													
Set_Conv_Sec_User_ID	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–													
Set_Conv_Type	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–													
Set_Deallocate_Type	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Set_Error_Direction	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	–
Set_Fill	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	–
Set_Initialization_Data	[pc]	↓	[sc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–		–											
Set_Log_Data	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	–
Set_Mapped_ Initialization_Data	[ok,pc,mp]	↓	[sc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–		–											
Set_Mode_Name	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–													
Set_Partner_ID	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–													
Set_Partner_LU_Name	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–													
Set_Prep_Data_Permitted	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Set_Prep_To_Rcv_Type	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Set_Processing_Mode	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	–
Set_Q_Callback_Func(N)	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	↓
[ok,pc]		–													–
Set_Q_Callback_Func(Q)	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Set_Q_Proc_Mode(N)	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	↓
[ok,pc]		–													–

Table 69 (Page 6 of 6). States and Transitions for CPI Communications Calls on Half-Duplex Conversations															
Set_Q_Proc_Mode(Q)	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Set_Receive_Type	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	–
Set_Return_Control	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–													
Set_Send_Rcv_Mode	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–													
Set_Send_Type	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	[sc]
[ok,pc]		–	–	–	–	–	–	–	–	–	–	–	–	–	
Set_Sync_Level	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc,pn]		–													
Set_TP_Name	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	↓
[ok,pc]		–													–
Set_Transaction_Control	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–													
Test_Req_To_Send_Rcd	[pc]	[sc]	↓′	↓′	↓′	[sc]	[sc]	[sc]	↓′	↓′	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]			–	–	–				–	–					
Wait_For_Completion	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok]	/	%	%	%	%	%	%	%	%	%	%	%	%	%	%
[pc]	/	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Wait_For_Conversation	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok]	/	%	%	%	%	%	%	%	%	%	%	%	%	%	%
[sc,se]	/	–	–	–	–	–	–	–	–	–	–	–	–	–	–

[9] The state table entries for Deferred_Deallocate and Include_Partner_In_Transaction calls are for conversations using an OSI TP CRM only. These calls get the CM_PROGRAM_PARAMETER_CHECK if issued on a conversation using an LU 6.2 CRM, regardless of the state.

[10] CPI Communications suspends action on the Deferred_Deallocate call until the transaction is committed or backed out.

[11] While the Initialize_Conversation call can be issued only once for any given conversation, a program can issue multiple Initialize_Conversation calls to establish concurrent conversations with different partners. For more information, see "Multiple Conversations" on page 30.

[12] The Request_To_Send call is not allowed in **Send, Send-Pending, Confirm-Send, Confirm-Deallocate, Sync-Point, Sync-Point-Send** or **Sync-Point-Deallocate**, or **Prepared** state when the conversation is using an OSI TP CRM. The call gets the CM_PROGRAM_STATE_CHECK return code.

Effects of Calls to the SAA Resource Recovery Interface on Half-Duplex Conversations

Table 70 shows the state transitions resulting from calls to the SAA resource recovery interface on half-duplex conversations. This table applies only to conversations with *sync_level* set to CM_SYNC_POINT.

The following abbreviations are used for return codes in Table 70:

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

Table 70. States and Transitions for Protected Half-Duplex Conversations (CPIRR)

Inputs	Reset 1	Initialize 2	Send 3	Receive 4	Send-Pending 5	Confirm 6	Confirm-Send 7	Confirm-Deallocate 8	Defer-Receive 9	Defer-Deallocate 10	Sync-Point 11	Sync-Point Send 12	Sync-Point Deallocate 13	Prepared 18	Initialize_Incoming 14
Commit call	[sc][13]	↓[14]	↓	[sc]	↓	[sc]	[sc]	↓[14]	↓	↓	↓	↓	↓	↓	↓[14]
[ok,com,cop]		–	–[15]		3			–	4	1	4	3	1	3[16]	–
[bo,bom,bop]		–	#		#			–	#	#	#	#	#	#	–
[sc]		–	–		–			–	–	–	–	–	–	–	–
Backout call	↓[17]	↓[14]	↓	↓	↓	↓	↓	↓[14]	↓	↓	↓	↓	↓	↓	↓[14]
[ok,bop,bom]	–	–	#	#	#	#	#	–	#	#	#	#	#	#	–

13 When a program started by an incoming conversation startup request issues a Commit call before issuing an Accept_Conversation call, a state check results. The Commit call has no effect on other conversations in **Reset** state.

14 Conversations in **Initialize**, **Initialize-Incoming**, or **Confirm-Deallocate** state are not affected by Commit and Backout calls.

15 The conversation goes to **Reset** state if the local program had issued a Deferred_Deallocate call prior to issuing the Commit call.

16 The conversation goes to **Reset** or **Receive** state if the local program had issued a Deferred_Deallocate call or a Prepare_To_Receive call with *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, respectively, prior to entering **Prepared** state.

17 When a program started by an incoming conversation startup request issues a Backout call before issuing an Accept_Conversation call, the underlying conversation is actually backed out, though the CPI Communications conversation remains in **Reset** state.

Effects of Calls on Half-Duplex Conversations to the X/Open TX Interface

Table 71 shows the state transitions resulting from calls to the X/Open TX interface on half-duplex conversations. This table applies only to conversations with *sync_level* set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM.

The following abbreviations are used for return codes in Table 71:

Abbreviation	Meaning
bo	TX_ROLLBACK or TX_ROLLBACK_NO_BEGIN
bom	TX_ROLLBACK_MIXED or TX_ROLLBACK_MIXED_NO_BEGIN
bop	TX_ROLLBACK_HAZARD or TX_ROLLBACK_HAZARD_NO_BEGIN
com	TX_MIXED or TX_MIXED_NO_BEGIN
cop	TX_HAZARD or TX_HAZARD_NO_BEGIN
fa	TX_FAIL
ok	TX_OK or TX_NO_BEGIN
sc	TX_PROTOCOL_ERROR

Table 71. States and Transitions for Protected Half-Duplex Conversations (X/Open TX)

Inputs	Reset 1	Initialize 2	Send 3	Receive 4	Send-Pending 5	Confirm 6	Confirm-Send 7	Confirm-Deallocate 8	Defer-Receive 9	Defer-Deallocate 10	Sync-Point 11	Sync-Point Send 12	Sync-Point Deallocate 13	Prepared 18	Initialize_Incoming 14
tx_commit	[sc][18]	↓[19]	↓	[sc]	↓	[sc]	[sc]	↓[19]	↓	↓	↓	↓	↓	↓	↓[19]
[ok,com,cop]		–	–[20]		3			–	4	1	4	3	1	3[21]	–
[bo,bom,bop]		–	#		#			–	#	#	#	#	#	#	–
[sc]		–	–		–			–	–	–	–	–	–	–	–
[fa]		–	?		?			–	?	?	?	?	?	?	–
tx_rollback	↓[22]	↓[19]	↓	↓	↓	↓	↓	↓[19]	↓	↓	↓	↓	↓	↓	↓[19]
[ok,bop,bom]	–	–	#	#	#	#	#	–	#	#	#	#	#	#	–
[fa]	–	–	?	?	?	?	?	–	?	?	?	?	?	?	–
tx_begin	–	–	–	–	–	–	–	–	–	–	/	/	/	/	–
tx_close	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
tx_info	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
tx_open	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
tx_set_commit_return	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
tx_set_trans_control	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
tx_set_trans_timeout	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

[18] When a program started by an incoming conversation startup request issues a tx_commit call before issuing an Accept_Conversation call, a state check results. The tx_commit call has no effect on other conversations in **Reset** state.

[19] Conversations in **Initialize**, **Initialize-Incoming**, or **Confirm-Deallocate** state are not affected by tx_commit and tx_rollback calls.

[20] The conversation goes to **Reset** state if the local program had issued a Deferred_Deallocate call prior to issuing the tx_commit call.

[21] The conversation goes to **Reset** or **Receive** state if the local program had issued a Deferred_Deallocate call or a Prepare_To_Receive call with *prepare_to_receive_type* set to CM_PREP_TO_RECEIVE_SYNC_LEVEL, respectively, prior to entering **Prepared** state.

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

Explanation of Full-Duplex State Table Abbreviations

Abbreviations are used in the state table to indicate the different permutations of calls and characteristics. There are four categories of abbreviations:

- **Conversation characteristic** abbreviations are enclosed by parentheses— (. . .)
- **Conversation queue** abbreviations are enclosed by parentheses— (. . .)
- **return_code** abbreviations are enclosed by brackets — [. . .]
- **data_received and status_received** abbreviations are enclosed by braces and separated by a comma— { . . . , . . . }. The abbreviation before the comma represents the *data_received* value, and the abbreviation after the comma represents the value of *status_received*.

The next sections show the abbreviations used in each category.

Conversation Characteristics ( )

The following abbreviations are used for conversation characteristics:

Abbreviation	Meaning
A	*deallocate_type* is set to CM_DEALLOCATE_ABEND
B	*send_type* is set to CM_BUFFER_DATA
C	*deallocate_type* is set to CM_DEALLOCATE_CONFIRM
D(x)	*send_type* is set to CM_SEND_AND_DEALLOCATE. *x* represents the *deallocate_type* and can be A, C, F, or S. Refer to the appropriate entries in this table for a description of these values.
F	For a Deallocate call, F means one of the following: • *deallocate_type* is set to CM_DEALLOCATE_FLUSH • *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and either *sync_level* is set to CM_NONE or the conversation is in **Initialize_Incoming** state • *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL and *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM, but the conversation is not currently included in a transaction For a Send_Data call, F means the following: • *send_type* is set to CM_SEND_AND_FLUSH
I	*receive_type* is set to CM_RECEIVE_IMMEDIATE
S	For a Deallocate call, S means the following: • *deallocate_type* is set to CM_DEALLOCATE_SYNC_LEVEL, *sync_level* is set to CM_SYNC_POINT_NO_CONFIRM and the conversation is currently included in a transaction
W	*receive_type* is set to CM_RECEIVE_AND_WAIT.

Conversation Queues ()

The following abbreviations are used for conversation queues:

Abbreviation	Meaning
N	*conversation_queue* is set to CM_INITIALIZATION_QUEUE
Q	*conversation_queue* is set to one of the following: • CM_SEND_QUEUE • CM_RECEIVE_QUEUE • CM_EXPEDITED_SEND_QUEUE • CM_EXPEDITED_RECEIVE_QUEUE

Return Code Values [ ]

The following table shows abbreviations that are used for return codes. The state table for CPI Communications calls on full-duplex conversations follows.

Abbreviation	Meaning
ae	For an Allocate call, ae means one of the following: • CM_ALLOCATE_FAILURE_NO_RETRY • CM_ALLOCATE_FAILURE_RETRY • CM_CONVERSATION_TYPE_MISMATCH • CM_PIP_NOT_SPECIFIED_CORRECTLY • CM_RESOURCE_FAILURE_NO_RETRY • CM_RESOURCE_FAILURE_RETRY • CM_RETRY_LIMIT_EXCEEDED • CM_SECURITY_MUTUAL_FAILED • CM_SECURITY_NOT_SUPPORTED • CM_SECURITY_NOT_VALID • CM_SEND_RCV_MODE_NOT_SUPPORTED • CM_SYNC_LVL_NOT_SUPPORTED_PGM • CM_SYNC_LVL_NOT_SUPPORTED_SYS • CM_TP_NOT_AVAILABLE_NO_RETRY • CM_TP_NOT_AVAILABLE_RETRY • CM_TPN_NOT_RECOGNIZED For any other call, ae means one of the following: • CM_ALLOCATION_ERROR • CM_CONVERSATION_TYPE_MISMATCH • CM_PIP_NOT_SPECIFIED_CORRECTLY • CM_SECURITY_NOT_VALID • CM_SEND_RCV_MODE_NOT_SUPPORTED • CM_SYNC_LVL_NOT_SUPPORTED_PGM • CM_SYNC_LVL_NOT_SUPPORTED_SYS • CM_TP_NOT_AVAILABLE_NO_RETRY • CM_TP_NOT_AVAILABLE_RETRY • CM_TPN_NOT_RECOGNIZED
bo	CM_TAKE_BACKOUT. This return code is returned only for conversations with *sync_level* set to CM_SYNC_POINT_NO_CONFIRM.
bs	CM_BUFFER_TOO_SMALL
da	da means one of the following: • CM_DEALLOCATED_ABEND • CM_DEALLOCATED_ABEND_SVC • CM_DEALLOCATED_ABEND_TIMER

Abbreviation	Meaning
db	db is returned only for conversations with *sync_level* set to CM_SYNC_POINT_NO_CONFIRM and means one of the following: • CM_DEALLOCATED_ABEND_BO • CM_DEALLOCATED_ABEND_SVC_BO • CM_DEALLOCATED_ABEND_TIMER_BO
dn	CM_DEALLOCATED_NORMAL
dr	CM_DEALLOCATE_CONFIRM_REJECT
ds	CM_CONV_DEALLOCATED_AFTER_SYNCPT. This return code is returned only for conversations with *sync_level* set to CM_SYNC_POINT_NO_CONFIRM.
ed	This return code is reported for expedited-data calls only. ed means one of the following: • CM_EXP_DATA_NOT_SUPPORTED • CM_CONVERSATION_ENDING
en	en means one of the following: • CM_PROGRAM_ERROR_NO_TRUNC • CM_SVC_ERROR_NO_TRUNC
mp	mp means one of the following: • CM_UNKNOWN_MAP_NAME_REQUESTED • CM_UNKNOWN_MAP_NAME_RECEIVED • CM_MAP_ROUTINE_ERROR
ep	ep means one of the following: • CM_PROGRAM_ERROR_PURGING • CM_SVC_ERROR_PURGING
et	et means one of the following: • CM_PROGRAM_ERROR_TRUNC • CM_SVC_ERROR_TRUNC
ns	CM_NO_SECONDARY_INFORMATION
oi	CM_OPERATION_INCOMPLETE
ok	CM_OK
pb	CM_INCLUDE_PARTNER_REJECT_BO
pc	CM_PROGRAM_PARAMETER_CHECK. This return code means an error was found in one or more parameters. For calls illegally issued in **Reset** state, pc is returned because the *conversation_ID* is undefined in that state.
pe	CM_PARAMETER_ERROR
pn	CM_PARM_VALUE_NOT_SUPPORTED
rb	rb means one of the following: • CM_RESOURCE_FAIL_NO_RETRY_BO • CM_RESOURCE_FAILURE_RETRY_BO
rf	rf means one of the following: • CM_RESOURCE_FAILURE_NO_RETRY • CM_RESOURCE_FAILURE_RETRY
sc	CM_PROGRAM_STATE_CHECK
un	CM_UNSUCCESSFUL

Notes:

1. The return code CM_PRODUCT_SPECIFIC_ERROR is not included in the state table because the state transitions caused by this return code are product-specific.
2. The CM_OPERATION_NOT_ACCEPTED return code is not included in the state table. A program receives CM_OPERATION_NOT_ACCEPTED when it issues a call associated with a queue that has a a previous operation still in progress, regardless of the state. No conversation state transition occurs.
3. The CM_CALL_NOT_SUPPORTED return code is not included in the state table. It is returned when the local system provides an entry point for the call but does not support the function requested by the call, regardless of the state. No state transition occurs.

data_received and status_received { , }

The following abbreviations are used for the *data_received* values:

Abbreviation	Meaning
dr	Means one of the following: • CM_DATA_RECEIVED • CM_COMPLETE_DATA_RECEIVED • CM_INCOMPLETE_DATA_RECEIVED
*	Means one of the following: • CM_DATA_RECEIVED • CM_COMPLETE_DATA_RECEIVED • CM_NO_DATA_RECEIVED

The following abbreviations are used for the *status_received* values:

Abbreviation	Meaning
cd	CM_CONFIRM_DEALLOCATE_RECEIVED
jt	CM_JOIN_TRANSACTION
no	CM_NO_STATUS_RECEIVED
po	CM_PREPARE_OK
tc	CM_TAKE_COMMIT or CM_TAKE_COMMIT_DATA_OK. These values are returned only for conversations with *sync_level* set to CM_SYNC_POINT_NO_CONFIRM.
td	CM_TAKE_COMMIT_DEALLOCATE or CM_TAKE_COMMIT_DEALLOC_DATA_OK. These values are returned only for conversations with *sync_level* set to CM_SYNC_POINT_NO_CONFIRM.

Table Symbols for the Full-Duplex State Table

The following symbols are used in the state table to indicate the condition that results when a call is issued from a certain state:

Symbol	Meaning
/	Cannot occur. CPI Communications either will not allow this input or will never return the indicated return codes for this input in this state.
–	Remain in current state
1-18	Number of next state
↓	It is valid to make this call from this state. See the table entries immediately below this symbol to determine the state transition resulting from the call.
%	Wait_for_Completion (CMWCMP) can only be issued when one or more of the calls issued by the program has received a *return code* of CM_OPERATION_INCOMPLETE. When Wait_for_Completion completes with a return code of CM_OK, it returns a list of outstanding-operation-IDs that identify the operations that have completed. Each of the conversations on which a call has completed then makes a transition to the appropriate state as indicated for the operations that are now complete.
↓′	For a conversation with *sync_level* set to CM_NONE or not currently included in a transaction, this is equivalent to ↓. If the conversation has *sync_level* set to CM_SYNC_POINT_NO_CONFIRM and the conversation is currently included in a transaction, however, ↓′ means it is valid to make this call from this state *unless the conversation's context is in the* **Backout-Required** *condition*. In that case, the call is invalid and CM_PROGRAM_STATE_CHECK is returned. For valid calls, see the table entries immediately below this symbol to determine the state transition resulting from the call.
^	For a conversation with *sync_level* set to CM_NONE or not currently included in a transaction, this symbol should be ignored. For a conversation using *sync_level* set to CM_SYNC_POINT_NO_CONFIRM and currently included in a transaction, when this symbol follows a state number or a – (for example, 1^ or –^), it means the conversation's context may be in the **Backout-Required** condition following the call.
#	Conversations with *sync_level* set to CM_SYNC_POINT_NO_CONFIRM go to the state they were in at the completion of the most recent synchronization point. If there was no prior synchronization event, both sides of the conversation go to **Send-Receive** state.

Note: The following calls can only be issued on half-duplex conversations. When issued on full-duplex conversations, CM_PROGRAM_PARAMETER_CHECK is returned for all conversation states except Reset. These calls are, therefore, not shown in the state tables.

- Confirm
- Prepare_To_Receive
- Request_To_Send
- Set_Confirmation_Urgency
- Set_Error_Direction
- Set_Prepare_To_Receive_Type
- Set_Processing_Mode
- Test_Request_To_Send_Received

Table 72 (Page 1 of 5). States and Transitions for CPI Communications Calls on Full-Duplex Conversations

Inputs	Used by CPI-C FDX conversations										
	Reset 1	Initialize 2	Confirm-Dealloc 8	Defer-Dealloc 10	Sync-Point 11	Sync-Pt-Dealloc 13	Initialize-Incoming 14	Send-Only 15	Receive-Only 16	Send-Receive 17	Prepared 18
Accept_Conversation	↓	/	/	/	/	/	/	/	/	/	/
[ok]	17										
[da,sc]	–										
Accept_Incoming	[pc]	[sc]	[sc]	[sc]	[sc]	[sc]	↓	[sc]	[sc]	[sc]	[sc]
[ok]							17				
[da]							1				
[oi,pc,sc]							–				
Allocate	[pc]	↓′	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]		17									
[ae]		1									
[oi,pc,pe,sc,un]		–									
Cancel_Conversation	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok]		1^	1^	1^	1^	1^	1^	1^	1^	1^	1^
[pc]		–	–	–	–	–	–	–	–	–	–
Confirmed[24]	[pc]	[sc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok]			1								
[oi,pc]			–								
Deallocate(A)	[pc]	↓	↓	↓	↓	↓	/	↓	↓	↓	↓
[ok]		1^	1^	1^	1^	1^		1^	1^	1^	1^
[oi,pc]		–	–	–	–	–		–	–	–	–
Deallocate(C)[24]	[pc]	[sc]	[sc]	[sc]	[sc]	[sc]	/	[sc]	[sc]	↓	[sc]
[ok,ae,da,dn,rf]										16	
[dr,oi,pc,sc]										–	
Deallocate(F)	[pc]	[sc]	[sc]	[sc]	[sc]	[sc]	↓	↓	[sc]	↓	[sc]
[ok]							1	1		16	
[ae]							/	/		16	
[da,dn,rf]							/	1		16	
[oi,pc]							–	–		–	
[sc]							/	/		–	
Deallocate(S)	[pc]	[sc]	[sc]	[sc]	[sc]	[sc]	/	/	/	↓′	[sc]
[ok]										10	
[ae,oi,pc,sc]										–	
[bo]										–^	
[db,pb,rb]										1^	
[ds]										1	
Deferred_Deallocate[25]	[pc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	↓′	[sc]
[ok]										–[26]	
[ae]										–	
[bo]										–^	
[db,pb,rb]										1^	
[oi,pc]										–	
Extract_AE_Qualifier	[pc]	↓	↓	↓	↓	↓	[sc]	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–		–	–	–	–
Extract_AP_Title	[pc]	↓	↓	↓	↓	↓	[sc]	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–		–	–	–	–
Extract_Appl_Ctx_Name	[pc]	↓	↓	↓	↓	↓	[sc]	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–		–	–	–	–
Extract_Conv_Context	[pc]	[sc]	↓	↓	↓	↓	[sc]	↓	↓	↓	↓
[ok,pc]			–	–	–	–		–	–	–	–
Extract_Conv_State	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–
[bo]		/	/	–	–	–	/	/	/	–	–
Extract_Conv_Type	[pc]	↓	↓	↓	↓	↓	[sc]	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–		–	–	–	–

Extract_Init_Data	[pc]	[sc]	↓	↓	↓	↓	[sc]	↓	↓	↓	↓
[ok,pc]			–	–	–	–		–	–	–	–
Extract_Mapped_ Initialization_Data	[pc]	[sc]	↓	↓	↓	↓	[sc]	↓	↓	↓	↓
[ok,pc,mp,bs]			–	–	–	–		–	–	–	–
Extract_Mode_Name	[pc]	↓	↓	↓	↓	↓	[sc]	↓	↓	↓	↓
[ok,pc,]		–	–	–	–	–		–	–	–	–
Extract_Part_LU_Name	[pc]	↓	↓	↓	↓	↓	[sc]	↓	↓	↓	↓
[ok,bs,pc,]		–	–	–	–	–		–	–	–	–
Extract_Partner_ID	↓	↓	↓	↓	↓	↓	[sc]	↓	↓	↓	↓
[ok,pc]	–	–	–	–	–	–		–	–	–	–
Extract_Sec_User_ID	[sc]	↓	↓	↓	↓	↓	[sc]	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–		–	–	–	–
Extract_Secondary_Info	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,ns,pc]	–	–	–	–	–	–	–	–	–	–	–
Extract_Send_Receive_Mode	[pc]	↓	↓	↓	↓	↓	[sc]	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–		–	–	–	–
Extract_Sync_Level	[pc]	↓	↓	↓	↓	↓	[sc]	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–		–	–	–	–
Extract_Transaction_Control	[pc]	↓	↓	↓	↓	↓	[sc]	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–		–	–	–	–
Extract_TP_Name	[pc]	↓	↓	↓	↓	↓	[sc]	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–		–	–	–	–
Flush	[pc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	↓	[sc]	↓′	[sc]
[ok]								–		–	
[ae]								/		16	
[bo]								/		–^	
[da,dn,rf]								1		16	
[db,rb]								/		1^	
[ds]								/		1	
[oi,pc]								–		–	
[pb]								/		1^	
[sc]								/		–	
Include_Ptr_In_Trans[25]	[pc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	↓′	[sc]
[ok]										–	
[ae]										–	
[da,dn,rf]										16	
[oi,pc]										–	
Initialize_Conversation[27]	↓	/	/	/	/	/	/	/	/	/	/
[ok]	2										
[pc]	–										
Initialize_for_Incoming	↓	/	/	/	/	/	/	/	/	/	/
[ok]	17										
[pc]	–										
Prepare	[pc]	[sc]	[sc]	↓′	[sc]	[sc]	[sc]	[sc]	[sc]	↓′	[sc]
[ok]				18						18	
[ae]				–						–	
[bo]				–^						–^	
[db,pb,rb]				1^						1^	
[ds]				1						1	
[oi,pc]				–						–	
Receive(I)	[pc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	↓	↓′	↓′
[ok] {dr,no}									–	–	–
[ok] {*,cd}									/	8	/
[ok] {*,jt}									/	–	/
[ok] {*,po}									/	/	–
[ok] {*,tc}									/	11	/
[ok] {*,td}									/	13	/
[ae]									1	1	1
[bo]									/	–^	–^

[da,rf]									1	1	/
[db,pb,rb]									/	1^	1^
[dn]									1	15	/
[ds]									/	1	/
[en,et]									–	–	17
[ep]									–	–	/
[pc,un]									–	–	–
[sc]									/	–	–
Receive_Mapped_Data(I)	[pc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	↓	↓′	↓′
[ok,mp] {dr,no}									–	–	–
[ok] {*,cd}									/	8	/
[ok] {*,jt}									/	–	/
[ok] {*,po}									/	/	–
[ok] {*,tc}									/	11	/
[ok] {*,td}									/	13	/
[ae]									1	1	1
[bo]									/	–^	–^
[da,rf]									1	1	/
[db,pb,rb]									/	1^	1^
[dn]									1	15	/
[ds]									/	1	/
[en,et]									–	–	17
[ep]									–	–	/
[pc,un]									–	–	–
[sc]									/	–	–
Receive(W)	[pc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	↓	↓′	↓′
[ok] {dr,no}									–	–	–
[ok] {*,cd}									/	8	/
[ok] {*,jt}									/	–	/
[ok] {*,po}									/	/	–
[ok] {*,tc}									/	11	/
[ok] {*,td}									/	13	/
[ae]									1	1	1
[bo]									/	–^	–^
[da,rf]									1	1	/
[db,pb,rb]									/	1^	1^
[dn]									1	15	/
[ds]									/	1	/
[en,et]									–	–	17
[ep]									–	–	/
[oi,pc]									–	–	–
[sc]									/	–	–
Receive_Mapped_Data(W)	[pc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	↓	↓′	↓′
[ok] {dr,no}									–	–	–
[ok] {*,cd}									/	8	/
[ok] {*,jt}									/	–	/
[ok] {*,po}									/	/	–
[ok] {*,tc}									/	11	/
[ok] {*,td}									/	13	/
[ae,mp]									1	1	1
[bo]									/	–^	–^
[da,rf]									1	1	/
[db,pb,rb]									/	1^	1^
[dn]									1	15	/
[ds]									/	1	/
[en,et]									–	–	17
[ep]									–	–	/
[oi,pc]									–	–	–
[sc]									/	–	–
Receive_Exp_Data	[pc]	[sc]	↓	↓	↓	↓	[sc]	↓	↓	↓	↓

[ok,ed,oi,pc,un]			–	–	–	–		–	–	–	–
Send_Data	[pc]	[sc]	[sc]	[sc]	↓′	↓′	[sc]	↓	[sc]	↓′	[sc]
(B) [ok]					–	–		–		–	
(D(A)) [ok]					1^	1^		1		1^	
(D(C)) [ok]					/	/		/		16	
(D(F)) [ok]					/	/		1		16	
(D(S)) [ok]					/	/		/		10	
(F) [ok]					–	–		–		–	
[ae]					/	/		/		16	
[bo]					–^	–^		/		–^	
[da,dn,rf]					/	/		1		16	
[db,rb]					1^	1^		/		1^	
[dr,oi,pc]					–	–		–		–	
[ds]					/	/		/		1	
[pb]					/	/		/		1^	
[sc]					–	–		/		–	
Send_Error	[pc]	[sc]	↓	[sc]	[sc]	[sc]	[sc]	↓	[sc]	↓′	[sc]
[ok]			17					–		–	
[ae]			/					/		16	
[bo]			/					/		–^	
[da,dn,rf]			1					1		16	
[db,rb]			/					/		1^	
[ds]			/					/		1	
[oi,pc]			–					–		–	
[pb]			/					/		1^	
[sc]			/					/		–	
Send_Expedited_Data	[pc]	[sc]	↓	↓	↓	↓	[sc]	↓	↓	↓	↓
[ok,ed,oi,pc]			–	–	–	–		–	–	–	–
Set_AE_Qualifier	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–									
Set_Allocate_Confirm	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–									
Set_AP_Title	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–									
Set_Appl_Context_Name	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–									
Set_Begin_Transaction	[pc]	↓	↓	↓	↓	↓	[sc]	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–		–	–	–	–
Set_Conv_Sec_PW	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–									
Set_Conv_Sec_Type	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc,pn]		–									
Set_Conv_Sec_User_ID	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–									
Set_Conversation_Type	[pc]	↓	[sc]	[sc]	[sc]	[sc]	↓	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–					–				
Set_Deallocate_Type	[pc]	↓	↓	↓	↓	↓	[sc]	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–		–	–	–	–
Set_Fill	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–
Set_Initialization_Data	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	↓	[sc]
[ok,pc]		–								–	
Set_Log_Data	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–
Set_Mapped_ Initialization_Data	[ok,pc,mp]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	↓	[sc]
[ok,pc]		–								–	
Set_Mode_Name	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–									
Set_Partner_LU_Name	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]

[ok,pc]		–									
Set_Partner_ID	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–									
Set_Prep_Data_Permitted	[pc]	↓	↓	↓	↓	↓	[sc]	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–
Set_Q_Callback_Func(N)	[pc]	↓	[sc]	[sc]	[sc]	[sc]	↓	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–					–				
Set_Q_Callback_Func(Q)	[pc]	↓	↓	↓	↓	↓	[sc]	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–		–	–	–	–
Set_Q_Proc_Mode(N)	[pc]	↓	[sc]	[sc]	[sc]	[sc]	↓	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–					–				
Set_Q_Proc_Mode(Q)	[pc]	↓	↓	↓	↓	↓	[sc]	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–		–	–	–	–
Set_Receive_Type	[pc]	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–	–	–	–	–	–
Set_Return_Control	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–									
Set_Send_Receive_Mode	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–									
Set_Send_Type	[pc]	↓	↓	↓	↓	↓	[sc]	↓	↓	↓	↓
[ok,pc]		–	–	–	–	–		–	–	–	–
Set_Sync_Level	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc,pn]		–									
Set_TP_Name	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	↓
[ok,pc]		–									
										–	
Set_Transaction_Control	[pc]	↓	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]	[sc]
[ok,pc]		–									
Wait_For_Completion	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
[ok]	/	%	%	%	%	%	%	%	%	%	%
[pc]	/	–	–	–	–	–	–	–	–	–	–

23 This state is entered only if the conversation is allocated using an OSI TP CRM.

24 This call can be issued only if the conversation is allocated using an OSI TP CRM.

25 The state table entries for Deferred_Deallocate and Include_Partner_In_Transaction calls are for conversations using an OSI TP CRM only. These calls get the CM_PROGRAM_PARAMETER_CHECK if issued on a conversation using an LU 6.2 CRM, regardless of the state.

26 CPI Communications suspends action on the Deferred_Deallocate call until the transaction is committed or backed out.

27 While the Initialize_Conversation call can be issued only once for any given conversation, a program can issue multiple Initialize_Conversation calls to establish concurrent conversations with different partners. For more information, see “Multiple Conversations” on page 30.

Effects of Calls to the SAA Resource Recovery Interface on Full-Duplex Conversations

Table 73 shows the state transitions resulting from calls to the SAA resource recovery interface on full-duplex conversations. This table applies only to conversations with *sync_level* set to CM_SYNC_POINT_NO_CONFIRM.

Commit and Backout are resource recovery calls. Their return codes are as follows.

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

Table 73. States and Transitions for Protected Full-Duplex Conversations (CPIRR)

	Used by CPI-C FDX conversations										
Inputs	**Reset 1**	**Initialize 2**	**Confirm-Dealloc 8**	**Defer-Dealloc 10**	**Sync-Point 11**	**Sync-Pt-Dealloc 13**	**Initialize-Incoming 14**	**Send-Only 15**	**Receive-Only 16**	**Send-Receive 17**	**Prepared 18**
Commit call[28]	[sc]	↓[29]	↓[29]	↓	↓	↓	↓[29]	[sc]	[sc]	↓	↓
[ok,com,cop]	–	–		1	17	1	–			–[30]	17[31]
[bo,bom,bop]		–	–	17	17	17	–			–	17
[fa,sc]		–	–	–	–	–	–			–	–
Backout call[32]	↓	↓[29]	↓[29]	↓	↓	↓	↓[29]	[sc]	[sc]	↓	↓
[bo,bom,bop]	–	–		17	17	17	–			–	17

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

[29] Conversations in **Initialize**, **Initialize-Incoming**, or **Confirm-Deallocate** state are not affected by Commit and Backout calls.

[30] The conversation goes to **Reset** state if the local program had issued a Deferred_Deallocate call prior to issuing the Commit call.

[31] The conversation goes to **Reset** state if the local program had issued a Deferred_Deallocate call prior to entering **Prepared** state.

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

Effects of Calls on Full-Duplex Conversations to the X/Open TX Interface

Table 74 shows the state transitions resulting from calls to the X/Open TX interface on full-duplex conversations. This table applies only to conversations with *sync_level* set to CM_SYNC_POINT_NO_CONFIRM.

The following abbreviations are used for return codes in Table 74:

Abbreviation	Meaning
bo	TX_ROLLBACK or TX_ROLLBACK_NO_BEGIN
bom	TX_ROLLBACK_MIXED or TX_ROLLBACK_MIXED_NO_BEGIN
bop	TX_ROLLBACK_HAZARD or TX_ROLLBACK_HAZARD_NO_BEGIN
com	TX_MIXED or TX_MIXED_NO_BEGIN
cop	TX_HAZARD or TX_HAZARD_NO_BEGIN
fa	TX_FAIL
ok	TX_OK or TX_NO_BEGIN
sc	TX_PROTOCOL_ERROR

Table 74. States and Transitions for Protected Full-Duplex Conversations (X/Open TX)

	Used by CPI-C FDX conversations										
Inputs	**Reset 1**	**Ini-tialize 2**	**Confirm-Dealloc 8**	**Defer-Dealloc 10**	**Sync-Point 11**	**Sync-Pt-Dealloc 13**	**Ini-tialize-Incoming 14**	**Send-Only 15**	**Receive-Only 16**	**Send-Receive 17**	**Prepared 18**
tx_commit	[sc][33]	↓[34]	↓[34]	↓	↓	↓	↓[34]	[sc]	[sc]	↓	↓
[ok,com,cop]		–	–	1	17	1	–			–[35]	17[36]
[bo,bom,bop]		–	–	17	17	17	–			–	17
[fa]		–	–	–	–	–	–			–	–
[sc]		–	–	–	–	–	–			–	–
tx_rollback	↓[37]	↓[34]	↓[34]	↓	↓	↓	↓[34]	[sc]	[sc]	↓	↓
[bo,bom,bop]	–	–		17	17	17	–			–	17
[fa]		–		–	–	–	–			–	–
tx_begin	–	–	/	–	–	/	–	–	–	–	/
tx_close	–	–	–	–	–	–	–	–	–	–	–
tx_info	–	–	–	–	–	–	–	–	–	–	–
tx_open	–	–	–	–	–	–	–	–	–	–	–
tx_set_commit_return	–	–	–	–	–	–	–	–	–	–	–
tx_set_trans_control	–	–	–	–	–	–	–	–	–	–	–
tx_set_trans_timeout	–	–	–	–	–	–	–	–	–	–	–

33 When a program started by an incoming conversation startup request issues a tx_commit call before issuing an Accept_Conversation call, a state check results. The tx_commit call has no effect on other conversations in **Reset** state.

34 Conversations in **Initialize**, **Initialize-Incoming**, or **Confirm-Deallocate** state are not affected by tx_commit and tx_rollback calls.

35 The conversation goes to **Reset** state if the local program had issued a Deferred_Deallocate call prior to issuing the tx_commit call.

36 The conversation goes to **Reset** state if the local program had issued a Deferred_Deallocate call prior to entering **Prepared** state.

37 When a program started by an incoming conversation startup request issues a tx_rollback call before issuing an Accept_Conversation call, the underlying conversation is actually backed out, though the CPI Communications conversation remains in **Reset** state.

Appendix D. CPI Communications and LU 6.2

This appendix is intended for programmers who are familiar with the LU 6.2 application programming interface. (LU 6.2 is also known as Advanced Program-to-Program Communication or APPC.) It describes the functional relationship between the APPC "verbs" and the CPI Communications calls described in this manual.

The CPI Communications calls have been built on top of the LU 6.2 verbs described in *SNA Transaction Programmer's Reference Manual for LU Type 6.2*. Table 75 beginning on page 729 shows the relationship between APPC verbs and CPI Communications calls. Use this table to determine how the function of a particular LU 6.2 verb is provided through CPI Communications.

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

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

- The Initialize_Conversation call and side information, used to initialize conversation characteristics without requiring the application program to explicitly specify these parameters.
- A conversation state of **Send-Pending** (discussed in more detail in "Send-Pending State and the error_direction Characteristic" on page 726).
- The Accept_Conversation call for use by a remote program to explicitly establish a conversation, the conversation identifier, and the conversation's characteristics.
- The *error_direction* conversation characteristic (discussed in more detail in "Send-Pending State and the error_direction Characteristic" on page 726).
- A *send_type* conversation characteristic for use in combining functions (this function was available with LU 6.2 verbs, but the verbs had to be issued separately).
- The capability to return both data and conversation status on the same Receive call.

Note: CPI Communications does *not* support the USER_CONTROL_DATA function that is available with the LU 6.2 interface:

To increase portability between systems, the character sets used to specify the partner *TP_name*, *partner_LU_name*, and *log_data* have been modified slightly from the character sets allowed by LU 6.2. To answer specific questions of compatibility, check the character sets described in Appendix A, "Variables and Characteristics."

Note: For mapping to OSI TP, see the CPI-C 2.1 Specification (SC31-6180-01).

© Copyright IBM Corp. 1996, 1998

Send-Pending State and the error_direction Characteristic

The **Send-Pending** state and *error_direction* characteristic are used in CPI Communications to eliminate ambiguity about the source of some errors. A program using CPI Communications can receive data and a change-of-direction indication at the same time. This "double function" creates a possibly ambiguous error condition, since it is impossible to determine whether a reported error (from Send_Error) was encountered because of the received data or after the processing of the change of direction.

The ambiguity is eliminated in CPI Communications by use of the **Send-Pending** state and *error_direction* characteristic. CPI Communications places the conversation in **Send-Pending** state whenever the program has received data and a *status_received* parameter of CM_SEND_RECEIVED (indicating a change of direction). Then, if the program encounters an error, it uses the Set_Error_Direction call to indicate how the error occurred. If the conversation is in **Send-Pending** state and the program issues a Send_Error call, CPI Communications examines the *error_direction* characteristic and notifies the partner program accordingly:

- If *error_direction* is set to CM_RECEIVE_ERROR, the partner program receives a *return_code* of CM_PROGRAM_ERROR_PURGING. This indicates that the error at the remote program occurred in the data, before (in LU 6.2 terms) the change-direction indicator was received.
- If *error_direction* is set to CM_SEND_ERROR, the partner program receives a *return_code* of CM_PROGRAM_ERROR_NO_TRUNC. This indicates that the error at the remote program occurred in the send processing after the change-direction indicator was received.

For an example of how CPI Communications uses the **Send-Pending** state and the *error_direction* characteristic, see "Example 7: Error Direction and Send-Pending State" on page 82.

Can CPI Communications Programs Communicate with APPC Programs?

Programs written using CPI Communications can communicate with APPC programs. Some examples of the limitations on the APPC program are:

- CPI Communications does not support PIP data.
- CPI Communications does not allow the specification of MAP_NAME.
- CPI Communications does not allow the specification of User_Control_Data.
- APPC programs with names containing characters no longer allowed may require a name change. See "SNA Service Transaction Programs" for a discussion of naming conventions for service transaction programs.

SNA Service Transaction Programs

If a CPI Communications program wants to specify an SNA service transaction program, the character set shown for *TP_name* in Appendix A, "Variables and Characteristics" is inadequate. The first character of an SNA service transaction program name is a character with a value in the range from `X'00'` through `X'0D'` or `X'10'` through `X'3F'` (excluding `X'0E'` and `X'0F'`). Refer to *SNA Transaction Programmer's Reference Manual for LU Type 6.2* for more details on SNA service transaction programs.

A CPI Communications program that has the appropriate privilege may specify the name of an SNA service transaction program for its partner *TP_name*. **Privilege** is an identification that a product or installation defines in order to differentiate LU service transaction programs from other programs, such as application programs. *TP_name* cannot specify an SNA service transaction program name at the mapped conversation protocol boundary.

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

Implementation Considerations

If the CRM used by the CPI-C implementation does not support full-duplex conversations, and the implementer desires to provide simulated full-duplex support, then the implementation should follow the *Simulated FDX Interoperability Recommendation* in order to interoperate with existing full-duplex simulations. The *Simulated FDX Interoperability Recommendation* is available on the Internet at URL:

`ftp://networking.raleigh.ibm.com/pub/standards/ciw/spec/fdxsim.psbin`

Relationship between LU 6.2 Verbs and CPI Communications Calls

Table 75 beginning on page 729 shows LU 6.2 verbs and their parameters on the left side and CPI Communications calls across the top. The table relates a verb or verb parameter to a call (not a call to a verb). A letter at the intersection of a verb or verb parameter row and a call column is interpreted as follows:

D This parameter has been set to a default value by the CPI Communications call. Default values can be found in the individual call descriptions.

X A similar or equal function for the LU 6.2 verb or parameter is available from the CPI Communications call. If more than one X appears on a line for a verb, the function is available by issuing a combination of the calls.

S This parameter can be set using the CPI Communications call.

Table 75. Relationship of LU 6.2 Verbs to CPI Communications Calls (Part 1)

CPI Communications Calls	**MC_ALLOCATE**	- LU_NAME	- MODE_NAME	- TPN	- TYPE	- RETURN_CONTROL	- CONVERSATION_GROUP_ID	- SYNC_LEVEL	- SECURITY	- PIP	**MC_CONFIRM**	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED	**MC_CONFIRMED**	**MC_DEALLOCATE**	- TYPE	- EXPEDITED_DATA_RECEIVED	**MC_FLUSH**	**MC_GET_ATTRIBUTES**	- PARTNER_LU_NAME	- PART_FULL_QUAL_LU_NAME	- MODE_NAME	- SYNC_LEVEL	- CONVERSATION_STATE	- CONV_CORRELATOR	- SESSION_ID	- CONVERSATION_GROUP_ID
Starter Set																											
Accept_Conversation																D											
Allocate	X																										
Deallocate															X												
Initilize_Conversation	X	D	D	D	D	D		D	D	D						D											
Receive																											
Send_Data																											
Advanced Function																											
Accept_Incoming																											
Cancel_Conversation															X	D											
Confirm											X	X	X														
Confirmed														X													
Extract_Conversation_State																			X					X			
Extract_Conversation_Type																											
Extract_Initialization_Data																											
Extract_Mapped_Initialization_Data																											
Extract_Mode_Name																			X			X					
Extract_Partner_LU_Name																			X	X	X						
Extract_Security_User_ID																											
Extract_Send_Receive_Mode																											
Extract_Sync_Level																			X				X				
Extract_TP_Name																											
Flush																		X									
Initialize_For_Incoming																D											
Prepare																											
Prepare_To_Receive																											
Receive_Expedited_Data																											
Receive_Mapped_Data																											
Request_To_Send																											
Send_Error																											
Send_Expedited_Data																											
Send_Mapped_Data																											
Set_Confirmation_Urgency																											
Set_Conversation_Security_Password	X								S																		
Set_Conversation_Security_Type	X								S																		
Set_Conversation_Security_User_ID	X								S																		
Set_Conversation_Type	X																										
Set_Deallocate_Type															X	S											
Set_Error_Direction																											
Set_Fill																											
Set_Initialization_Data	X									S																	
Set_Mapped_Initialization_Data	X									S																	
Set_Log_Data																											
Set_Mode_Name	X		S																								
Set_Partner_LU_Name	X	S																									
Set_Prepare_To_Receive_Type																											
Set_Receive_Type																											
Set_Return_Control	X					S																					
Set_Send_Receive_Mode	X				S																						
Set_Send_Type																											
Set_Sync_Level	X							S																			
Set_TP_Name	X			S																							
Test_Request_To_Send_Received																											

Table 76. Relationship of LU 6.2 Verbs to CPI Communications Calls (Part 2)

CPI Communications Calls	**MC_POST_ON_RECEIPT**	**MC_PREPARE_FOR_SYNCPT**	**MC_PREPARE_TO_RECEIVE**	- TYPE	- LOCKS	**MC_RECEIVE_AND_WAIT**	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED	- WHAT_RECEIVED	- MAP_NAME	**MC_RCV_EXPEDITED_DATA**	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED	**MC_RECEIVE_IMMEDIATE**	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED	- WHAT_RECEIVED	- MAP_NAME	**MC_REQUEST_TO_SEND**	**MC_SEND_DATA**	- MAP_NAME	- USER_CONTROL_DATA	- ENCRYPT	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED
Starter Set																									
Accept_Conversation				D	D																				
Allocate																									
Deallocate																									
Initilize_Conversation				D	D																				
Receive						X	X	X	X					X	X	X	X								
Send_Data																				X				X	X
Advanced Function																									
Accept_Incoming																									
Cancel_Conversation																									
Confirm																									
Confirmed																									
Extract_Conversation_State																									
Extract_Conversation_Type																									
Extract_Mapped_Initialization_Data																									
Extract_Mode_Name																									
Extract_Partner_LU_Name																									
Extract_Security_User_ID																									
Extract_Send_Receive_Mode																									
Extract_Sync_Level																									
Extract_TP_Name																									
Flush																									
Initialize_For_Incoming				D	D																				
Prepare		X																							
Prepare_To_Receive			X																						
Receive_Expedited_Data											X	X	X												
Receive_Mapped_Data						X	X	X	X	X				X	X	X	X	X							
Request_To_Send																			X						
Send_Error																									
Send_Expedited_Data																									
Send_Mapped_Data																				X	X			X	X
Set_Confirmation_Urgency			X		S																				
Set_Conversation_Security_Password																									
Set_Conversation_Security_Type																									
Set_Conversation_Security_User_ID																									
Set_Conversation_Type																									
Set_Deallocate_Type																									
Set_Error_Direction																									
Set_Fill																									
Set_Initialization_Data																									
Set_Log_Data																									
Set_Mapped_Initialization_Data																									
Set_Mode_Name																									
Set_Partner_LU_Name																									
Set_Prepare_To_Receive_Type			X	S																					
Set_Receive_Type						X								X											
Set_Return_Control																									
Set_Send_Receive_Mode																									
Set_Send_Type																									
Set_Sync_Level																									
Set_TP_Name																									
Test_Request_To_Send_Received																									

Table 77. Relationship of LU 6.2 Verbs to CPI Communications Calls (Part 3)

CPI Communications Calls	**MC_SEND_ERROR**	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED	**MC_SEND_EXPEDITED_DATA**	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED	**MC_TEST**	- TEST_POSTED	- TEST_REQ_TO_SEND_RCVD	**BACKOUT**	**GET_TP_PROPERTIES**	- OWN_FULLY_QUAL_LU_NAME	- OWN_TP_NAME	- OWN_TP_INSTANCE	- SECURITY_USER_ID	- SECURITY_PROFILE	- LUW_IDENTIFIER	- PROTECTED_LUW_IDENTIFIER	**GET_TYPE**	**SET_SYNCPT_OPTIONS**	**SYNCPT**	- REQ_TO_SEND_RECEIVED	**WAIT**	- RESOURCE_POSTED	**WAIT_FOR_COMPLETION**
Starter Set																									
Accept_Conversation																									
Allocate																									
Deallocate																									
Initilize_Conversation																									
Receive																									
Send_Data																									
Advanced Function																									
Accept_Incoming																									
Cancel_Conversation																									
Confirm																									
Confirmed																									
Extract_Conversation_State																									
Extract_Conversation_Type																			X						
Extract_Mode_Name																									
Extract_Partner_LU_Name																									
Extract_Security_User_ID											X				X										
Extract_Send_Receive_Mode																			X						
Extract_Sync_Level																									
Extract_TP_Name											X		X												
Flush																									
Initialize_For_Incoming																									
Prepare																									
Prepare_To_Receive																									
Receive_Expedited_Data																									
Request_To_Send																									
Send_Error	X	X	X																						
Send_Expedited_Data				X	X	X																			
Set_Confirmation_Urgency																									
Set_Conversation_Security_Password																									
Set_Conversation_Security_Type																									
Set_Conversation_Security_User_ID																									
Set_Conversation_Type																									
Set_Deallocate_Type																									
Set_Error_Direction	X																								
Set_Fill																									
Set_Initialization_Data																									
Set_Log_Data																									
Set_Mode_Name																									
Set_Partner_LU_Name																									
Set_Prepare_To_Receive_Type																									
Set_Receive_Type																									
Set_Return_Control																									
Set_Send_Receive_Mode																									
Set_Send_Type																									
Set_Sync_Level																									
Set_TP_Name																									
Test_Request_To_Send_Received							X		X																

Table 78. Relationship of LU 6.2 Verbs to CPI Communications Calls (Part 4)

CPI Communications Calls	**ALLOCATE**	- LU_NAME	- MODE_NAME	- TPN	- TYPE	- RETURN_CONTROL	- CONVERSATION_GROUP_ID	- SYNC_LEVEL	- SECURITY	- PIP	**CONFIRM**	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED	**CONFIRMED**	**DEALLOCATE**	- TYPE	- LOG_DATA	- EXPEDITED_DATA_RECEIVED	**FLUSH**
Starter Set																			
Accept_Conversation																D	D		
Allocate	X																		
Deallocate															X				
Initilize_Conversation	X	D	D	D	D	D		D	D	D						D	D		
Receive																			
Send_Data																			
Advanced Function																			
Accept_Incoming																			
Cancel_Conversation																			
Confirm											X	X	X						
Confirmed														X					
Extract_Conversation_State																			
Extract_Conversation_Type																			
Extract_Mode_Name																			
Extract_Partner_LU_Name																			
Extract_Security_User_ID																			
Extract_Send_Receive_Mode																			
Extract_Sync_Level																			
Extract_TP_Name																			
Flush																			X
Initialize_For_Incoming																D	D		
Prepare																			
Prepare_To_Receive																			
Receive_Expedited_Data																			
Request_To_Send																			
Send_Error																			
Send_Expedited_Data																			
Set_Confirmation_Urgency																			
Set_Conversation_Security_Password	X								S										
Set_Conversation_Security_Type	X								S										
Set_Conversation_Security_User_ID	X								S										
Set_Conversation_Type	X				S														
Set_Deallocate_Type															X	S			
Set_Error_Direction																			
Set_Fill																			
Set_Initialization_Data	X									S									
Set_Log_Data															X		S		
Set_Mode_Name	X		S																
Set_Partner_LU_Name	X	S																	
Set_Prepare_To_Receive_Type																			
Set_Receive_Type																			
Set_Return_Control	X					S													
Set_Send_Receive_Mode	X				S														
Set_Send_Type																			
Set_Sync_Level	X							S											
Set_TP_Name	X			S															
Test_Request_To_Send_Received																			

Table 79. Relationship of LU 6.2 Verbs to CPI Communications Calls (Part 5)

CPI Communications Calls	**GET_ATTRIBUTES**	- PARTNER_LU_NAME	- PART_FULL_QUAL_LU_NAME	- MODE_NAME	- SYNC_LEVEL	- CONVERSATION_STATE	- CONV_CORRELATOR	- SESSION_ID	- CONVERSATION_GROUP_ID	**POST_ON_RECEIPT**	- FILL	**PREPARE_FOR_SYNCPT**	**PREPARE_TO_RECEIVE**	- TYPE	- LOCKS	**RECEIVE_AND_WAIT**	- FILL	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED	- WHAT_RECEIVED	**RECEIVE_EXPEDITED_DATA**	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED
Starter Set																							
Accept_Conversation														D	D		D						
Allocate																							
Deallocate																							
Initilize_Conversation														D	D		D						
Receive																X		X	X	X			
Send_Data																							
Advanced Function																							
Accept_Incoming																							
Cancel_Conversation																							
Confirm																							
Confirmed																							
Extract_Conversation_State	X					X																	
Extract_Conversation_Type																							
Extract_Mode_Name	X			X																			
Extract_Partner_LU_Name	X	X	X																				
Extract_Security_User_ID																							
Extract_Send_Receive_Mode																							
Extract_Sync_Level	X				X																		
Extract_TP_Name																							
Flush																							
Initialize_For_Incoming														D	D		D						
Prepare												X											
Prepare_To_Receive													X										
Receive_Expedited_Data																					X	X	X
Request_To_Send																							
Send_Error																							
Send_Expedited_Data																							
Set_Confirmation_Urgency													X		S								
Set_Conversation_Security_Password																							
Set_Conversation_Security_Type																							
Set_Conversation_Security_User_ID																							
Set_Conversation_Type																							
Set_Deallocate_Type																							
Set_Error_Direction																							
Set_Fill																X	S						
Set_Initialization_Data																							
Set_Log_Data																							
Set_Mode_Name																							
Set_Partner_LU_Name																							
Set_Prepare_To_Receive_Type													X	S									
Set_Receive_Type																X							
Set_Return_Control																							
Set_Send_Receive_Mode																							
Set_Send_Type																							
Set_Sync_Level																							
Set_TP_Name																							
Test_Request_To_Send_Received																							

Table 80. Relationship of LU 6.2 Verbs to CPI Communications Calls (Part 6)

CPI Communications Calls	**RECEIVE_IMMEDIATE**	- FILL	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED	- WHAT_RECEIVED	**REQUEST_TO_SEND**	**SEND_DATA**	- ENCRYPT	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED	**SEND_EXPEDITED_DATA**	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED	**SEND_ERROR**	- TYPE	- LOG_DATA	- REQ_TO_SEND_RECEIVED	- EXPEDITED_DATA_RECEIVED	**TEST**	- TEST_POSTED	- TEST_REQ_TO_SEND_RCVD
Starter Set																					
Accept_Conversation		D														D					
Allocate																					
Deallocate																					
Initilize_Conversation		D														D					
Receive	X		X	X	X																
Send_Data							X		X	X											
Advanced Function																					
Accept_Incoming																					
Cancel_Conversation																					
Confirm																					
Confirmed																					
Extract_Conversation_State																					
Extract_Conversation_Type																					
Extract_Mode_Name																					
Extract_Partner_LU_Name																					
Extract_Security_User_ID																					
Extract_Send_Receive_Mode																					
Extract_Sync_Level																					
Extract_TP_Name																					
Flush																					
Initialize_For_Incoming		D															D				
Prepare																					
Prepare_To_Receive																					
Receive_Expedited_Data																					
Request_To_Send						X															
Send_Error														X	D		X	X			
Send_Expedited_Data											X	X	X								
Set_Confirmation_Urgency																					
Set_Conversation_Security_Password																					
Set_Conversation_Security_Type																					
Set_Conversation_Security_User_ID																					
Set_Conversation_Type																					
Set_Deallocate_Type																					
Set_Error_Direction														X							
Set_Fill	X	S																			
Set_Initialization_Data																					
Set_Log_Data														X		S					
Set_Mode_Name																					
Set_Partner_LU_Name																					
Set_Prepare_To_Receive_Type																					
Set_Receive_Type	X																				
Set_Return_Control																					
Set_Send_Receive_Mode																					
Set_Send_Type																					
Set_Sync_Level																					
Set_TP_Name																					
Test_Request_To_Send_Received																			X		X

Appendix E. Application Migration from X/Open CPI-C

This appendix describes the application migration from the original X/Open CPI-C (as defined in the 1990 edition of *X/Open Developers' Specification CPI-C*) to CPI-C 2.0. The following section lists the differences between X/Open CPI-C and CPI-C 2.0 that may require that changes be made to move a program from X/Open CPI-C to CPI-C 2.0.

Application Migration Considerations

- When the Accept_Conversation call is issued and no incoming conversation exists, CPI-C 2.0 returns CM_PROGRAM_STATE_CHECK; X/Open CPI-C returns CM_OPERATION_INCOMPLETE and a conversation identifier. Programs that want to accept multiple conversations using the non-blocking processing mode should use the following calls instead of Accept_Conversation:
  - Initialize_For_Incoming
  - Set_Processing_Mode, with *processing_mode* set to CM_NON_BLOCKING
  - Accept_Incoming.

  See “Example 13: Accepting Multiple Conversations Using Conversation-Level Non-Blocking Calls” on page 94 for accepting conversations using non-blocking Accept_Incoming calls.
- CPI-C 2.0 returns the CM_OPERATION_NOT_ACCEPTED return code if a conversation call is issued and the previous call on the conversation has not completed. X/Open CPI-C returns CM_PROGRAM_STATE_CHECK.
- The Extract_Conversation_Security_User_ID (CMECSU) call of X/Open CPI-C is not supported in CPI-C 2.0. The function is available in CPI-C 2.0 using the Extract_Security_User_ID (CMESUI) call, which supports an increased length of the *security_user_id*.
- The CPI-C 2.0 Convert_Incoming and Convert_Outgoing calls return CM_PROGRAM_PARAMETER_CHECK if the string_length exceeds the maximum length permitted by the local implementation. This return code is not supported in X/Open CPI-C for these calls.
- The CM_SYNC_LEVEL_NOT_SUPPORTED_PGM return code of X/Open CPI-C must be changed to CM_SYNC_LVL_NOT_SUPPORTED_PGM.
- The specify_Local_TP_Name call of X/Open CPI-C must be changed to Specify_Local_TP_Name.
- X/Open CPI-C allows several processes to share the same conversation identifier. In CPI-C 2.0, the scope of the *conversation_ID* is system-dependent.
- X/Open defines all functions as type CM_RETCODE (for example, extern CM_RETCODE CMACCP;). CPI-C 2.0 defines all functions as type void (for example, extern CM_ENTRY cmaccp;). Programs that test the return value of an X/Open CPI-C call have to test the *return_code* parameter value in CPI-C 2.0.

© Copyright IBM Corp. 1996, 1998

- The readable macros (for example, Accept_Conversation instead of CMACCP) are only available in CPI-C 2.0 if the program or the include file contains a line

```
#define READABLE_MACROS
```

- Some parameter types in X/Open CPI-C and CPI-C 2.0 differ, as shown in Table 81. Some compilers may give out warnings when compiling existing X/Open CPI-C programs with the new CPI-C 2.0 include file.

Table 81. Parameter Type Differences

Parameter Type	**X/Open CPI-C**	**CPI-C 2.0**
Conversation ID Parameters	CONVERSATION_ID (that is, char [8])	unsigned char CM_PTR (that is, unsigned char *)
Character Pointers	char *	unsigned char CM_PTR (that is, unsigned char *)
Length Parameters	int *	CM_INT32 CM_PTR (that is, signed long int *)
Definitions of Return Codes and Numeric Parameters	typedef enum	#define

Appendix F. CPI Communications Extensions for Use with DCE Directory

This appendix describes CPI Communications extensions in two areas:

- Additional directory objects (beyond program installation object) to provide additional functionality. Three new objects are introduced to allow enhanced program use of the directory at a functional level equivalent to Distributed Computing Environment (DCE) Remote Procedure Call (RPC) use. The three new objects are:
  - **Profile** object, accessed by a profile distinguished name (PDN)
  - **Server** object, accessed by a server distinguished name (SDN)
  - **Server Group** object, accessed by a server group distinguished name (SGDN)
- A CPI Communications-specific interface for interaction with the DCE directory, referred to here as the CPI-C name service interface (CNSI).

The rest of this appendix describes the three new objects, the CNSI, and how they can be used to interact with CPI Communications.

Profile Object

The profile object enables the establishment of a search priority for all of the other directory objects. For example, a typical profile might define a primary server for first attempts in locating function, but also define a secondary server if the primary server is unable.

The structure of the profile object is a list of profile elements, where each element contains three parts:

- PFID—the program function identifier for the profile element. This field can be null if a default ordering for all function is desired.
- Priority—the priority of the element relative to all other profile elements.
- Distinguished Name (DN)—points to either a profile, server, server-group, or program-installation object.

Server Object

Conceptually, the server object represents a collection of programs and resources; examples are a single system or a file server on a LAN.

The server object enables two key capabilities:

1. Location/search for a particular function. For example, is OfficeNetMail application available on this server?

   This function is provided by maintaining a list of (PFID, PIDN) pairs in the server object, where the PIDN is a DN for the program installation object that provides the function of the PFID. If a function is available from more than one program installation on the server, there are multiple (PFID, PIDN) pairs.

© Copyright IBM Corp. 1996, 1998

2. Location/search of a particular resource. For example, does the PAYROLL DATABASE reside on this server?

   This function is provided by maintaining a list of resources that are available on the server.

Server Group Object

The server group object extends the concept of a server by allowing servers to be grouped. Grouping provides such additional value as transparent redundancy and the ability to specify equivalent servers.

The server group object pointed to by an SGDN simply contains a list of server distinguished names (SDNs).

Interaction of Directory Objects

Figure 32 shows a simple diagram of the different directory objects and their interactions. The arrows show the interconnection of information between different objects.

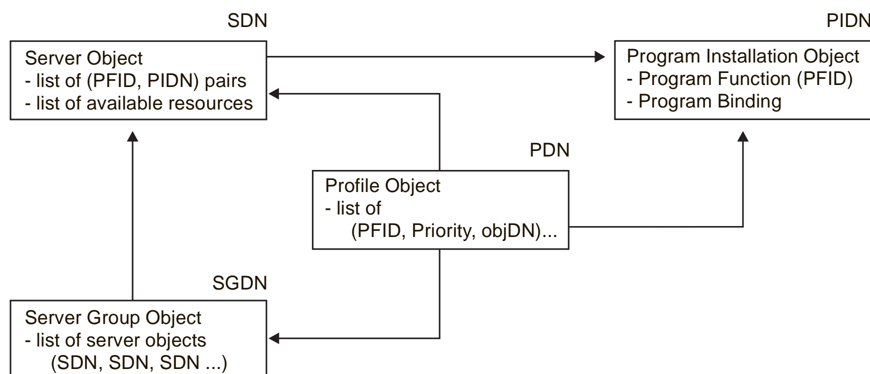


Figure 32. Interactions of Directory Objects

Here is an example of how a program might use the objects:

- The program specifies a profile object, a PFID, and a resource.
- The profile object is accessed first to determine servers that have the requested function. By using the priority field, a prioritized list of relevant servers is obtained.
- The first server object in the list is accessed. The resource list in the server object is then searched to determine if the server has the requested resource. If the resource is not found, the next server object in the list is accessed.
- Once a server with the correct resource is located, the (PFID, PIDN) list is used to determine the DN for the program installation object.
- The program installation object is accessed to obtain the program binding.

CPI-C Name Service Interface

The objects described above reside in the distributed directory and are accessed by a directory interface. For example, programs using the DCE distributed directory can access directory objects using the X/OPEN Directory Services (XDS) and X/OPEN Object Management (XOM) programming interfaces. Because these interfaces are not specific to the CPI Communications directory objects, this Appendix describes a local service, referred to here as the **CPI-C name service interface** (CNSI), which programs can use to interact with CPI Communications directory objects in the directory.

Figure 33 shows the interaction between the program, CPI-C, and CNSI.

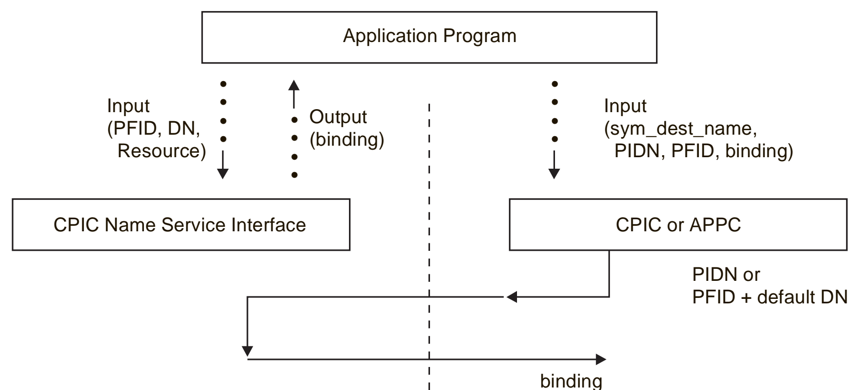


Figure 33. CPI-C Name Service Interface (CNSI) and CPI Communications

The right side of the figure shows how a program makes calls to CPI Communications using the function described in Chapter 2, “CPI Communications Terms and Concepts.”

If the program specifies a sym_dest_name pointing to a DN, or a PFID or DN directly, CPI Communications accesses the distributed directory and acquires a program binding. CPI-C uses the program binding to establish a conversation with the partner program. When a PFID is specified, a default DN is used to begin the search. This default DN may be either a PDN, SDN, or SGDN. Definition of a default DN is system specific.

The left side of Figure 33 shows how a program uses the CNSI and objects defined in this Appendix to locate the partner program. The application program makes a call to the CNSI and specifies a three-parameter search criteria:

- **program function ID**—a PFID that represents the function the partner program provides.
- **priority/location**—a DN representing either a profile, server, or server group object.
- **resource**—a resource_ID identifying a particular resource (that is, a file that is required along with a particular function).

CNSI Calls

As previously discussed, the program uses CNSI calls to retrieve a PIDN, which it then passes to CPI-C using the Set_Partner_ID call. Although CNSI calls are not CPI Communications calls and, thus, not formally defined, the interface would look very much like DCE's NSI interface for RPC. Here is a sample listing of the type of calls that would be required to manipulate the objects.

Table 82. Sample Calls for CNSI

cpic_ns_binding_export	cpic_ns_group_mbr_inq_done
cpic_ns_binding_import_begin	cpic_ns_group_mbr_inq_next
cpic_ns_binding_import_done	cpic_ns_group_mbr_remove
cpic_ns_binding_import_next	cpic_ns_mgmt_binding_unexport
cpic_ns_binding_inq_entry_name	cpic_ns_mgmt_entry_create
cpic_ns_binding_lookup_begin	cpic_ns_mgmt_entry_delete
cpic_ns_binding_lookup_done	cpic_ns_mgmt_entry_inq_if_ids
cpic_ns_binding_lookup_next	cpic_ns_mgmt_handle_set_exp_age
cpic_ns_binding_select	cpic_ns_mgmt_inq_exp_age
cpic_ns_binding_unexport	cpic_ns_mgmt_set_exp_age
cpic_ns_entry_expand_name	cpic_ns_profile_delete
cpic_ns_entry_object_inq_begin	cpic_ns_profile_elt_add
cpic_ns_entry_object_inq_done	cpic_ns_profile_elt_inq_begin
cpic_ns_entry_object_inq_next	cpic_ns_profile_elt_inq_done
cpic_ns_group_delete	cpic_ns_profile_elt_inq_next
cpic_ns_group_mbr_add	cpic_ns_profile_elt_remove
cpic_ns_group_mbr_inq_begin	cpic_ns_set_authn

Definition of New Objects

Three new objects are required:

- Profile
- Server
- Server Group

This section discusses terminology and then provides definitions for attributes in the new objects.

Terminology

The following terminology is used to define the new CNSI attributes:

- A **multi-valued** attribute means the directory stores (and is aware of) multiple values for the attribute.
- A **single-valued** attribute means the directory has only one value for the attribute.
- A **complex** attribute means the directory is only aware of a single value for the attribute, but the single-value is encoded to contain multiple fields. A complex field follows the generic coding guidelines provided in Appendix A, “Variables and Characteristics” for the program binding.
- A **simple** attribute means the attribute value is not encoded; that is, there is only one field.

Profile Object

The Profile object contains one attribute, a profile record attribute:

- profile record (complex, multi-valued)

  A value for the profile record has three parts: priority, PFID, DN

  1. priority is 2-character decimal digits (00=highest)
  2. PFID is a program function identifier (as defined in Appendix A, "Variables and Characteristics")
  3. DN is a distinguished name for one of the following:
     - Server object
     - Server group object
     - Profile object
     - Program installation object

Note: The profile object has the unique object identifier of 1.3.18.0.2.6.8. The profile record attribute has the unique object identifier of 1.3.18.0.2.4.15.

Server Object

The Server object contains two attributes, a server record and resource record attribute:

- Server record (complex, multi-valued)

  A value for the server record has two parts: PFID and DN.

  1. PFID is a program function identifier.
  2. DN is a distinguished name for a program installation object.

- Resource record (simple, multi-valued)

  A value for the resource record is a customer-defined byte string with a maximum length of 64 bytes.

Note: The server object has the unique object identifier of 1.3.18.0.2.6.9. The server record attribute has the unique object identifier of 1.3.18.0.2.4.16. The resource record attribute has the unique object identifier of 1.3.18.0.2.4.17.

Server Group Object

The Server Group object contains one attribute, a distinguished name attribute:

- distinguished name (simple, multi-valued)

  Valid DN values are distinguished names that represent:

  - Profile object
  - Server object
  - Server group object

Note: The server group object has the unique object identifier of 1.3.18.0.2.6.10. The distinguished name attribute is stored with an object identifier of 2.5.4.31.

Program Installation Object

The Program Installation object is defined by CPI Communications, but is included here for completeness. It contains two attributes, a program_function_ID and a program binding attribute:

- Program function ID (simple, single-valued)

  See Appendix A, "Variables and Characteristics" for a description.
- Program binding (complex, single-valued)

  See Appendix A, "Variables and Characteristics" for a description.

Encoding Method for Complex Attribute Values

The encoding method for complex attribute values is to follow the general method outlined in Appendix A, "Variables and Characteristics" for the program binding. Each complex value is a string of the general pattern "field_type=field_value; field_type=field_value" where field_type is a four-byte identifier for the field and field_value is a variable-length field. The end of a field_value is determined by the semicolon character. If the field_value itself contains a semicolon, the textual semicolon is repeated. Correct ordering of fields is shown above in the object definitions.

Table 83 shows the 4-byte code for the field types required in the complex attributes of the new objects, along with a maximum length for each field_value.

Table 83. Fields in Complex Attributes

Description of Field Contents	4-Byte field_type Code	Maximum Length of Field
priority—used in the profile record and server record attribute.	PRIO	2
program function ID—used in the profile record and server record attribute	PFID	1024
distinguished name—used in the profile record and server record attribute	DSTN	1024

Scenarios for Use of CNSI

Programs access the CNSI directly by specifying a tuple of (function, priority/location, resource) and retrieve a PIDN. This PIDN is then passed to CPI-C using the Set_Partner_ID call. The program is able to control retry because it has the explicit option of retrieving each successive PIDN and attempting the Allocate.

A sample sequence of calls is illustrated in Figure 34.

```
cpic_ns_binding_lookup_begin (
                PFID,                     (input)
                entry_name_syntax,        (input)
                entry_DN_to_start_search, (input)
                ResID,                    (input)
                ns_handle,                (output)
                status  )                 (output)

While unsucessful do
  begin

     cpic_ns_binding_lookup_next  (
                     ns_handle,  (input)
                     next_PIDN,       (output)
                     status  )   (output)

     CMINIT(sym_dest_name = '        ',
            conversation_id)

     CMSPID(conversation_ID,
           partner_ID_type = CM_DISTINGUISHED_NAME,
                partner_ID = next_PIDN,
         partner_ID_length = length(next_PIDN),
          partner_ID_scope = CM_EXPLICIT,
           return_code)

     CMALLC(conversation_id, return_code)

  end

  cpic_ns_done(ns_handle,         (input)  /* clear up control blocks   */
          return_code)       (output)
```

Figure 34. Program Uses CNSI to Locate PIDN

The sections that follow show the different combinations of parameters that can be provided to CNSI. The three-tuple shown as headings represents (program_function_ID, DN, resource_ID).

(PFID, *, *)

- Target: any program installation providing the function of the PFID.
- CNSI reads default profile using PFID, returns PIDNs.

(PFID, SDN, *)

- Target: any program installation within the server that provides the function specified by the PFID.
- CNSI reads SDN object, returns PIDNs paired with specified PFID.

(PFID, SGDN, *)

- Target: any program installation on any server within a server group.
- CNSI reads SGDN object, obtaining list of SDNs. CNSI reads SDN objects (in random order), returning PIDNs that were paired with the specified PFID.

(PFID, SDN, resID)

- Target: program installation on the server with the specified resource and function.
- CNSI reads SDN object and searches for match with resID. If match found, CNSI returns PIDNs that were paired with the specified PFID.

(PFID, SGDN, resID)

- Target: program installation on the server with the specified resource, within a server group.
- CNSI reads SGDN object, obtaining list of SDNs. CNSI reads SDN objects (in random order) and searches for match with resID. If match found, CNSI returns PIDNs that were paired with the specified PFID.

(PFID, PDN, *)

- Target: any program installation providing the function specified by the PFID, using the priority specified by the profile object.
- CNSI reads profile object and creates a list of elements with the specified PFID. In priority order, the associated object (PIDN, SDN, or SGDN) is read, and PIDNs are returned as above.

(PFID, PDN, resID)

- Target: program installation providing the specified function on a server with the specified resource, search order specified by profile object.
- CNSI reads profile object and orders the list of elements with the PFID. In priority order, the associated object (SDN or SGDN) is read, and PIDNs are returned as above. Note that any PIDN found in the profile would have no meaning in this case, as there would be no associated resource.

Appendix G. CPI Communications 2.1 Conformance Classes

The purpose of the CPI-C 2.1 conformance classes is to foster an orderly marketplace for CPI-C 2.1 implementation, purchase, and use. The conformance class definition assists the CPI-C implementer in deciding what to build, the CPI-C purchaser in knowing what to buy, and the CPI-C application builder in knowing what to use. Uses of the conformance class definition include the following:

- Product announcements
- Application requirements specifications
- Procurement specifications
- Conformance test suite development

Definitions

Mandatory conformance class
A function set that an implementation must support to conform to CPI-C. All of the function within the set must be implemented.

Optional conformance class
A function set that an implementation may support. To support an optional conformance class, an implementation must support all the function in the optional conformance class and in its prerequisite conformance classes.

Prerequisite conformance class
A conformance class required for support of another conformance class.

Conformance Requirements

To conform with CPI-C 2.1, an implementation must support the following:

- The mandatory conformance class (conversations)
- Either the LU 6.2 or the OSI TP conformance class, or both

Additionally, an implementation may support any optional conformance class.

Multi-Threading Support

While CPI Communications itself does not provide multi-threading support, some implementations are designed to work with multi-threading support in the base operating system and to allow multi-threaded programs to use CPI Communications. It is assumed that an implementation that supports multi-threaded programs allows concurrent operations through the use of multiple program threads. See “Concurrent Operations” on page 44.

CPI-C 2.1 Conformance Classes

CPI-C 2.1 conformance classes consist of a mandatory conformance class and a number of optional conformance classes. A conformance class may be a functional conformance class, consisting of function that may be supported by a CPI-C implementation, or a configuration conformance class, specifying how a CPI-C

© Copyright IBM Corp. 1996, 1998

implementation may allow an installation to configure the available support. The conformance classes are listed here and described below.

Functional conformance classes:

- Conversations (mandatory)
- LU 6.2
- OSI TP
- Recoverable transactions
- Unchained transactions
- Conversation-level non-blocking
- Queue-level non-blocking
- Callback function
- Server
- Data conversion routines
- Security
- Distributed security
- Full-duplex
- Expedited data
- Directory
- Secondary information
- Initialization data
- Automatic data conversion

Configuration conformance class:

- OSI TP addressing disable

Functional Conformance Class Descriptions

For each conformance class, the following information is provided:

- Brief description
- Prerequisite conformance classes
- Required calls
- Whether support is mandatory or optional

Additional details of the required support (characteristics, variables, and values) are included in the tables in “Conformance Class Details” on page 755.

Conversations

Description: allows a program to start and end half-duplex conversations, to exchange data on those converations, to use confirmation, error notification, the attention mechanism (Request_To_Send), and the optimization calls (Flush, Prepare_To_Receive), and to modify and examine conversation or system characteristics.

Prerequisites: none

Required Starter Set calls:

- CMACCP - Accept_Conversation
- CMALLC - Allocate
- CMDEAL - Deallocate
- CMINIT - Initialize_Conversation
- CMRCV - Receive

- CMSEND - Send_Data

Required Advanced Function calls:

- CMCFM - Confirm
- CMCFMD - Confirmed
- CMECS - Extract_Conversation_State
- CMECT - Extract_Conversation_Type
- CMEMBS - Extract_Maximum_Buffer_Size
- CMEMN - Extract_Mode_Name
- CMESL - Extract_Sync_Level
- CMFLUS - Flush
- CMPTR - Prepare_To_Receive
- CMRTS - Request_To_Send
- CMSERR - Send_Error
- CMSCT - Set_Conversation_Type
- CMSDT - Set_Deallocate_Type
- CMSF - Set_Fill
- CMSLD - Set_Log_Data
- CMSMN - Set_Mode_Name
- CMSPTR - Set_Prepare_To_Receive_Type
- CMSRT - Set_Receive_Type
- CMSRC - Set_Return_Control
- CMSST - Set_Send_Type
- CMSSL - Set_Sync_Level
  Required *sync_level* values:
  – CM_NONE
  – CM_CONFIRM
- CMSTPN - Set_TP_Name
- CMTRTS - Test_Request_To_Send_Received

Support: mandatory

LU 6.2

Description: allows a program to use LU 6.2-specific services.

Prerequisites: conversations

Required calls:

- CMEPLN - Extract_Partner_LU_Name
- CMSED - Set_Error_Direction
- CMSPLN - Set_Partner_LU_Name

Support: optional. However, an implementation must support either this conformance class or the OSI TP conformance class.

OSI TP

Description: allows a program to use OSI TP-specific services.

Prerequisites: conversations

Required calls:

- CMEAEQ - Extract_AE_Qualifier

- CMEAPT - Extract_AP_Title
- CMEACN - Extract_Application_Context_Name
- CMSAC - Set_Allocate_Confirm
- CMSAEQ - Set_AE_Qualifier
- CMSAPT - Set_AP_Title
- CMSACN - Set_Application_Context_Name
- CMSCU - Set_Confirmation_Urgency

Support: optional. However, an implementation must support either this conformance class or the LU 6.2 conformance class.

Recoverable Transactions

Description: allows a program to use CPI-C in conjunction with a resource recovery interface to coordinate changes to distributed resources using two-phase commit protocols.

Prerequisites: conversations

Required calls:

- CMSSL - Set_Sync_Level
  Required *sync_level* value:
  - CM_SYNC_POINT

  Required *sync_level* value if either full duplex or OSI TP is also supported:
  - CM_SYNC_POINT_NO_CONFIRM

Required calls if OSI TP is also supported:

- CMDFDE - Deferred_Deallocate
- CMPREP - Prepare
- CMSPDP - Set_Prepare_Data_Permitted

Required call if X/Open TX resource recovery interface is also supported:

- CMSJT - Set_Join_Transaction

Support: optional

Unchained Transactions

Description: allows a program to complete a recoverable transaction with a commit call without immediately starting a new recoverable transaction. The program may begin a new recoverable transaction at a later time.

Prerequisites: conversations, recoverable transactions, OSI TP

Required calls:

- CMETC - Extract_Transaction_Control
- CMINCL - Include_Partner_In_Transaction
- CMSBT - Set_Begin_Transaction
- CMSTC - Set_Transaction_Control

Support: optional

Conversation-Level Non-Blocking

Description: allows a program to regain control if a call cannot complete immediately. The call remains in progress. A program can have one outstanding operation on a conversation.

Prerequisites: conversations

Required calls:

- CMCANC - Cancel_Conversation
- CMSPM - Set_Processing_Mode
- CMWAIT - Wait_For_Conversation

Support: optional

Queue-Level Non-Blocking

Description: allows a program to regain control if a call cannot complete immediately. The call remains in progress. A program can have one outstanding operation per conversation queue.

Prerequisites: conversations

Required calls:

- CMCANC - Cancel_Conversation
- CMSQPM - Set_Queue_Processing_Mode
- CMWCMP - Wait_For_Completion

Support: optional

Callback Function

Description: allows a program to regain control if a call cannot complete immediately. The call remains in progress. A program can have one outstanding operation per conversation queue. A program receives an asynchronous notification when an outstanding operation completes.

Prerequisites: conversations

Required calls:

- CMCANC - Cancel_Conversation
- CMSQCF - Set_Queue_Callback_Function

Support: optional

Server

Description: allows a program to register multiple TP names with CPI-C, to accept multiple incoming conversations, and to manage contexts for different clients.

Prerequisites: conversations. Additionally, for a system that does not support multi-threaded programs: one or more of conversation-level non-blocking, queue-level non-blocking, or callback function.

Required calls:

- CMACCI - Accept_Incoming
- CMECTX - Extract_Conversation_Context
- CMETPN - Extract_TP_Name
- CMINIC - Initialize_For_Incoming
- CMRLTP - Release_Local_TP_Name
- CMSLTP - Specify_Local_TP_Name

Support: optional

Data Conversion Routines

Description: allows a program to call local routines to change the encoding of a character string from the local encoding to EBCDIC, or vice versa.

Prerequisites: conversations

Required calls:

- CMCNVI - Convert_Incoming
- CMCNVO - Convert_Outgoing

Support: optional

Security

Description: allows a program to establish conversations that use access security information provided administratively in side information or set directly by the program.

Prerequisites: conversations

Required calls:

- CMESUI - Extract_Security_User_ID
- CMSCSP - Set_Conversation_Security_Password
- CMSCST - Set_Conversation_Security_Type
  Required *conversation_security_type* values:
  - CM_SECURITY_NONE
  - CM_SECURITY_PROGRAM
  - CM_SECURITY_PROGRAM_STRONG
  - CM_SECURITY_SAME
- CMSCSU Set_Conversation_Security_User_ID

Support: optional

Distributed Security

Description: allows a program to use security services provided by a distributed security server.

Prerequisites: conversations, directory

Required calls:

- CMESUI - Extract_Security_User_ID
- CMSCST - Set_Conversation_Security_Type
  Required *conversation_security_type* values:
  - CM_SECURITY_NONE
  - CM_SECURITY_SAME
  - CM_SECURITY_DISTRIBUTED
  - CM_SECURITY_MUTUAL

Support: optional

Full-Duplex

Description: allows a program to use full-duplex conversations.

Prerequisites: conversations. Additionally, for a system that does not support multi-threaded programs: either queue-level non-blocking or callback function.

Required calls:

- CMESRM - Extract_Send_Receive_Mode
- CMSSRM - Set_Send_Receive_Mode

Support: optional

Expedited Data

Description: allows a program to exchange expedited data with the partner program.

Prerequisites: conversations, LU 6.2

Required calls:

- CMRCVX - Receive_Expedited_Data
- CMSNDX - Send_Expedited_Data

Support: optional

Directory

Description: allows a program to use destination information stored in a distributed directory.

Prerequisites: conversations

Required calls:

- CMEPID - Extract_Partner_ID
- CMSPID - Set_Partner_ID

Support: optional

Secondary Information

Description: allows a program to extract secondary information associated with the return code for a given call.

Prerequisites: conversations

Required calls:

- CMESI - Extract_Secondary_Information

Support: optional

Initialization Data

Description: allows a program to exchange initialization data at conversation start-up.

Prerequisites: conversations

Required calls:

- CMEID - Extract_Initialization_Data
- CMSID - Set_Initialization_Data

Support: mandatory if OSI TP is supported; otherwise, optional.

Automatic Data Conversation

Description: allows a program to use automatic data conversion on initialization data and data records sent and received.

Prerequisites: conversations

Required calls:

- CMRCVM - Receive_Mapped_Data
- CMSNDM - Send_Mapped_Data

Required calls if Initialization Data is also supported:

- CMEMID - Extract_Mapped_Initialization_Data
- CMSMID - Set_Mapped_Initialization_Data

Support: optional

Configuration Conformance Class Description

Configuration conformance classes specify configuration options that an implementation may support. This section describes the configuration conformance class. The following information is provided:

- Brief description
- Prerequisite conformance classes
- Required configuration options
- Whether support is mandatory or optional

OSI TP Addressing Disable

Description: an installation may disable the addressing Set calls for OSI TP. A program that issues a disabled call gets the CM_CALL_NOT_SUPPORTED return code.

Prerequisites: conversations, OSI TP

Calls that may be disabled:

- CMSAEQ - Set_AE_Qualifier
- CMSAPT - Set_AP_Title
- CMSACN - Set_Application_Context_Name

Support: optional

Relationship to OSI TP Functional Units and OSI TP Profiles

Table 84 shows the OSI TP service functional units and the corresponding CPI-C 2.1 conformance classes. For each functional unit, the calls in the corresponding conformance classes map to the services defined by the functional unit. For the Commit, Chained Transactions, and Unchained Transactions functional units, support for a resource recovery interface is also required.

Table 84. OSI TP Service Functional Units and Corresponding Conformance Classes

Service Functional Unit	Conformance Classes
Dialogue	conversations, OSI TP
Shared Control	full-duplex
Polarized Control	conversations
Handshake	conversations
Commit	recoverable transactions
Chained Transactions	recoverable transactions
Unchained Transactions	unchained transactions

Table 85 shows the OSI TP profiles defined by the OSI Implementers Workshop and the corresponding CPI-C 2.1 conformance classes. For each profile, the calls in the corresponding conformance classes map to the services defined by the profile. The protocol functional units included in each profile are shown (in parentheses). For the Commit, Chained Transactions, Unchained Transactions, and Recovery functional units, support for a resource recovery interface is also required.

Note: Profiles ATP12, ATP22, and ATP32 specify that the handshake functional unit is optional. CPI-C does not support the use of Confirm/Confirmed (handshake) on a full duplex conversation.

Table 85. OSI TP Profiles and Corresponding Conformance Classes

Profile (protocol functional units)	Conformance Classes
ATP11 (dialogue, handshake, polarized control)	conversations, OSI TP
ATP21 (dialogue, handshake, polarized control, commit, unchained transactions, recovery)	conversations, OSI TP, recoverable transactions, unchained transactions
ATP31 (dialogue, handshake, polarized control, commit, chained transactions, recovery)	conversations, OSI TP, recoverable transactions
ATP12 (dialogue, handshake (optional), shared control)	conversations, OSI TP, full-duplex
ATP22 (dialogue, handshake (optional), shared control, commit, unchained transactions, recovery)	conversations, OSI TP, full-duplex, recoverable transactions, unchained transactions
ATP32 (dialogue, handshake (optional), shared control, commit, chained transactions, recovery)	conversations, OSI TP, full-duplex, recoverable transactions

Conformance Class Details

This section provides additional information on the required support for each functional conformance class. The tables below list each CPI-C call, characteristic, variable, and value in the left column; any conformance class or combination of conformance classes that requires the implementation of that call, characteristic, variable, or value is listed in the right column. Combinations are enclosed in parentheses. For example, (OSI TP and recoverable transactions) to the right of the Deferred_Deallocate call indicates that the call is required if both the OSI TP and recoverable transactions conformance classes are supported.

The conformance class requirements for calls are presented in Table 86, and the conformance class requirements for characteristics, variables, and values are presented in Table 87 on page 757.

Table 86 (Page 1 of 3). Conformance Class Requirements—Calls

Call	Required by
CMACCP — Accept_Conversation	conversations
CMACCI — Accept_Incoming	server
CMALLC — Allocate	conversations
CMCANC — Cancel_Conversation	conversation-level non-blocking, callback function, queue-level non-blocking
CMCFM — Confirm	conversations
CMCFMD — Confirmed	conversations
CMCNVI — Convert_Incoming	data conversion routines
CMCNVO — Convert_Outgoing	data conversion routines
CMDEAL — Deallocate	conversations
CMDFDE — Deferred_Deallocate	(OSI TP and recoverable transactions)
CMEAEQ — Extract_AE_Qualifier	OSI TP
CMEAPT — Extract_AP_Title	OSI TP
CMEACN — Extract_Application_Context_Name	OSI TP
CMECTX — Extract_Conversation_Context	server
CMECS — Extract_Conversation_State	conversations
CMECT — Extract_Conversation_Type	conversations
CMEID — Extract_Initialization_Data	initialization data
CMEMBS — Extract_Maximum_Buffer_Size	conversations
CMEMID — Extract_Mapped_Initialization_Data	automatic data conversion
CMEMN — Extract_Mode_Name	conversations
CMEPB — Extract_Partner_ID	directory
CMEPLN — Extract_Partner_LU_Name	LU 6.2
CMESI — Extract_Secondary_Information	secondary information
CMESUI — Extract_Security_User_ID	security, distributed security
CMESRM — Extract_Send_Receive_Mode	full-duplex
CMESL — Extract_Sync_Level	conversations
CMETPN — Extract_TP_Name	conversations
CMETC — Extract_Transaction_Control	unchained transactions
CMFLUS — Flush	conversations
CMINCL — Include_Partner_In_Transaction	unchained transactions

Table 86 (Page 2 of 3). Conformance Class Requirements—Calls

Call	Required by
CMINIC — Initialize_For_Incoming	server
CMINIT — Initialize_Conversation	conversations
CMPREP — Prepare	(OSI TP and recoverable transactions)
CMPTR — Prepare_To_Receive	conversations
CMRCV — Receive	conversations
CMRCVM — Receive_Mapped_Data	automatic data conversion
CMRCVX — Receive_Expedited_Data	expedited data
CMRLTP — Release_Local_TP_Name	server
CMRTS — Request_To_Send	conversations
CMSEND — Send_Data	conversations
CMSERR — Send_Error	conversations
CMSNDM — Send_Mapped_Data	automatic data conversion
CMSNDX — Send_Expedited_Data	expedited data
CMSAEQ — Set_AE_Qualifier	OSI TP
CMSAC — Set_Allocate_Confirm	OSI TP
CMSAPT — Set_AP_Title	OSI TP
CMSACN — Set_Application_Context_Name	OSI TP
CMSBT — Set_Begin_Transaction	unchained transactions
CMSCU — Set_Confirmation_Urgency	OSI TP
CMSCSP — Set_Conversation_Security_Password	security
CMSCST — Set_Conversation_Security_Type	security, distributed security
CMSCSU — Set_Conversation_Security_User_ID	security
CMSCT — Set_Conversation_Type	conversations
CMSDT — Set_Deallocate_Type	conversations
CMSED — Set_Error_Direction	LU 6.2
CMSF — Set_Fill	conversations
CMSID — Set_Initialization_Data	initialization data
CMSJT — Set_Join_Transaction	recoverable transactions (if TX interface is supported)
CMSLD — Set_Log_Data	conversations
CMSMID — Set_Mapped_Initialization_Data	automatic data conversion
CMSMN — Set_Mode_Name	conversations
CMSPID — Set_Partner_ID	directory
CMSPLN — Set_Partner_LU_Name	LU 6.2
CMSPDP — Set_Prepare_Data_Permitted	(OSI TP and recoverable transactions)
CMSPTR — Set_Prepare_to_Receive_Type	conversations
CMSPM — Set_Processing_Mode	conversation-level non-blocking
CMSQCF — Set_Queue_Callback_Function	callback function
CMSQPM — Set_Queue_Processing_Mode	queue-level non-blocking
CMSRT — Set_Receive_Type	conversations
CMSRC — Set_Return_Control	conversations
CMSSRM — Set_Send_Receive_Mode	full-duplex
CMSST — Set_Send_Type	conversations
CMSSL — Set_Sync_Level	conversations

Table 86 (Page 3 of 3). Conformance Class Requirements—Calls

Call	Required by
CMSTPN — Set_TP_Name	conversations
CMSTC — Set_Transaction_Control	unchained transactions
CMSLTP — Specify_Local_TP_Name	server
CMTRTS — Test_Request_To_Send_Received	conversations
CMWCMP — Wait_For_Completion	queue-level non-blocking
CMWAIT — Wait_For_Conversation	conversation-level non-blocking

Table 87 (Page 1 of 5). Conformance Class Requirements—Characteristics, Variables, and Values

Characteristics, Variables, and Values	Required by
ae_qualifier	OSI TP
ae_qualifier_format cm_dn cm_oid	OSI TP OSI TP OSI TP
ae_qualifier_length	OSI TP
allocate_confirm cm_allocate_confirm cm_allocate_no_confirm	OSI TP OSI TP OSI TP
ap_title	OSI TP
ap_title_format cm_dn cm_oid	OSI TP OSI TP OSI TP
ap_title_length	OSI TP
application_context_name	OSI TP
application_context_name_length	OSI TP
begin_transaction cm_begin_explicit cm_begin_implicit	OSI TP OSI TP OSI TP
buffer	conversations
buffer_length	conversations
callback_function	callback function
call_ID	secondary information
completed_op_index_list	queue-level non-blocking
completed_op_count	queue-level non-blocking
confirmation_urgency cm_confirmation_not_urgent cm_confirmation_urgent	OSI TP OSI TP OSI TP
context_ID	server
context_ID_length	server
control_information_received cm_no_control_info_received cm_req_to_send_received cm_exp_data_received cm_rts_and_exp_data_received cm_allocate_confirmed cm_allocate_confirmed_with_data cm_allocate_rejected_with_data	conversations conversations conversations expedited data expedited data OSI TP OSI TP OSI TP
conversation_ID	conversations

Table 87 (Page 2 of 5). Conformance Class Requirements—Characteristics, Variables, and Values

Characteristics, Variables, and Values	Required by
conversation_queue	queue-level non-blocking, callback function
cm_initialization_queue	queue-level non-blocking, callback function
cm_send_queue	queue-level non-blocking, callback function
cm_receive_queue	queue-level non-blocking, callback function
cm_send_receive_queue	queue-level non-blocking, callback function
cm_expedited_send_queue	queue-level non-blocking, callback function
cm_expedited_receive_queue	queue-level non-blocking, callback function
conversation_return_code	conversation-level non-blocking
(same values as return_code)	
conversation_security_type	security, distributed security
cm_security_none	security, distributed security
cm_security_same	security, distributed security
cm_security_program	security
cm_security_program_strong	security
cm_security_distributed	distributed security
cm_security_mutual	distributed security
conversation_state	conversations
cm_initialize_state	conversations
cm_send_state	conversations
cm_receive_state	conversations
cm_send_pending_state	conversations
cm_confirm_state	conversations
cm_confirm_send_state	conversations
cm_confirm_deallocate_state	conversations
cm_defer_receive_state	recoverable transactions
cm_defer_deallocate_state	recoverable transactions
cm_sync_point_state	recoverable transactions
cm_sync_point_send_state	recoverable transactions
cm_sync_point_deallocate_state	recoverable transactions
cm_initialize_incoming_state	server
cm_prepared_state	recoverable transactions
cm_send_only_state	full-duplex
cm_receive_only_state	full-duplex
cm_send_and_receive_state	full-duplex
conversation_type	conversations
cm_basic_conversation	conversations
cm_mapped_conversation	conversations
data_received	conversations
cm_no_data_received	conversations
cm_data_received	conversations
cm_complete_data_received	conversations
cm_incomplete_data_received	conversations
deallocate_type	conversations
cm_deallocate_sync_level	conversations
cm_deallocate_flush	conversations
cm_deallocate_confirm	conversations
cm_deallocate_abend	conversations
directory_encoding	directory
cm_default_encoding	directory
cm_unicode_encoding	directory
directory_syntax	directory
cm_default_syntax	directory
cm_DCE_syntax	directory
cm_XDS_syntax	directory
cm_NDS_syntax	directory
error_direction	LU 6.2
cm_receive_error	LU 6.2
cm_send_error	LU 6.2

Table 87 (Page 3 of 5). Conformance Class Requirements—Characteristics, Variables, and Values

Characteristics, Variables, and Values	Required by
expedited_receive_type	expedited data
cm_receive_and_wait	expedited data
cm_receive_immediate	expedited data
fill	conversations
cm_fill_ll	conversations
cm_fill_buffer	conversations
initialization_data	OSI TP
initialization_data_length	OSI TP
join_transaction	recoverable transactions (if TX interface is supported)
cm_explicit	recoverable transactions (if TX interface is supported)
cm_implicit	recoverable transactions (if TX interface is supported)
log_data	conversations
log_data_length	conversations
map_name	automatic data conversion
map_name_length	automatic data conversion
maximum_buffer_size	conversations
mode_name	conversations
mode_name_length	conversations
OOID	queue-level non-blocking
OOID_list	queue-level non-blocking
OOID_list_count	queue-level non-blocking
partner_ID	directory
partner_ID_length	directory
partner_ID_scope	directory
cm_explicit	directory
cm_reference	directory
partner_ID_type	directory
cm_distinguished_name	directory
cm_local_distinguished_name	directory
cm_program_function_ID	directory
cm_OSI_TPSU_title_OID	directory
cm_program_binding	directory
partner_LU_name	LU 6.2
partner_LU_name_length	LU 6.2
prepare_data_permitted	(OSI TP and recoverable transactions)
cm_prepare_data_permitted	(OSI TP and recoverable transactions)
cm_prepare_data_not_permitted	(OSI TP and recoverable transactions)
prepare_to_receive_type	conversations
cm_prep_to_receive_sync_level	conversations
cm_prep_to_receive_flush	conversations
cm_prep_to_receive_confirm	conversations
processing_mode	conversation-level non-blocking
cm_blocking	conversation-level non-blocking
cm_non_blocking	conversation-level non-blocking
queue_processing_mode	queue-level non-blocking
cm_blocking	queue-level non-blocking
cm_non_blocking	queue-level non-blocking
receive_type	conversations
cm_receive_and_wait	conversations
cm_receive_immediate	conversations
received_length	conversations

Table 87 (Page 4 of 5). Conformance Class Requirements—Characteristics, Variables, and Values

Characteristics, Variables, and Values	Required by
requested_length	conversations
return_code	conversations
cm_ok	conversations
cm_allocate_failure_no_retry	conversations
cm_allocate_failure_retry	conversations
cm_conversation_type_mismatch	LU 6.2
cm_pip_not_specified_correctly	LU 6.2
cm_security_not_valid	conversations
cm_sync_lvl_not_supported_sys	conversations
cm_sync_lvl_not_supported_pgm	LU 6.2
cm_tpn_not_recognized	conversations
cm_tp_not_available_no_retry	conversations
cm_tp_not_available_retry	conversations
cm_deallocated_abend	conversations
cm_deallocated_normal	conversations
cm_parameter_error	conversations
cm_product_specific_error	conversations
cm_program_error_no_trunc	LU 6.2
cm_program_error_purging	conversations
cm_program_error_trunc	LU 6.2
cm_program_parameter_check	conversations
cm_program_state_check	conversations
cm_resource_failure_no_retry	conversations
cm_resource_failure_retry	conversations
cm_unsuccessful	conversations
cm_deallocated_abend_svc	LU 6.2
cm_deallocated_abend_timer	LU 6.2
cm_svc_error_no_trunc	LU 6.2
cm_svc_error_purging	LU 6.2
cm_svc_error_trunc	LU 6.2
cm_operation_incomplete	conversation-level non-blocking, callback function, queue-level non-blocking
cm_system_event	conversation-level non-blocking
cm_operation_not_accepted	conversation-level non-blocking, callback function, queue-level non-blocking
cm_conversation_ending	full-duplex
cm_send_rcv_mode_not_supported	OSI TP, full-duplex
cm_buffer_too_small	expedited data, OSI TP, directory
cm_exp_data_not_supported	expedited data
cm_deallocate_confirm_reject	(OSI TP and full-duplex)
cm_allocation_error	full-duplex
cm_retry_limit_exceeded	directory
cm_no_secondary_information	secondary information
cm_security_not_supported	security, distributed security
cm_security_mutual_failed	distributed security
cm_call_not_supported	-
cm_parm_value_not_supported	conversations, security, distributed security recoverable transactions
cm_unknown_map_name_requested	automatic data conversion
cm_unknown_map_name_received	automatic data conversion
cm_map_routine_error	automatic data conversion
cm_conversation_cancelled	conversation-level non-blocking, callback function, queue-level non-blocking
cm_take_backout	recoverable transactions
cm_deallocated_abend_bo	recoverable transactions
cm_deallocated_abend_svc_bo	(LU 6.2 and recoverable transactions)
cm_deallocated_abend_timer_bo	(LU 6.2 and recoverable transactions)
cm_resource_fail_no_retry_bo	recoverable transactions
cm_resource_failure_retry_bo	recoverable transactions
cm_deallocated_normal_bo	recoverable transactions
cm_conv_dealloc_after_syncpt	(full-duplex, recoverable transactions and LU 6.2)
cm_include_partner_reject_bo	unchained transactions

Table 87 (Page 5 of 5). Conformance Class Requirements—Characteristics, Variables, and Values

Characteristics, Variables, and Values	Required by
return_control	conversations
cm_when_session_allocated	conversations
cm_immediate	conversations
cm_when_conwinner_allocated	conversations
cm_when_session_free	conversations
security_password	security
security_password_length	security
security_user_ID	security, distributed security
security_user_ID_length	security, distributed security
send_length	conversations
send_receive_mode	full-duplex
cm_full_duplex	full-duplex
cm_half_duplex	full-duplex
send_type	conversations
cm_buffer_data	conversations
cm_send_and_flush	conversations
cm_send_and_confirm	conversations
cm_send_and_prep_to_receive	conversations
cm_send_and_deallocate	conversations
status_received	conversations
cm_no_status_received	conversations
cm_send_received	conversations
cm_confirm_received	conversations
cm_confirm_send_received	conversations
cm_confirm_dealloc_received	conversations
cm_take_commit	recoverable transactions
cm_take_commit_send	recoverable transactions
cm_take_commit_deallocate	recoverable transactions
cm_take_commit_data_permitted	(OSI TP and recoverable transactions)
cm_take_commit_send_data_permitted	(OSI TP and recoverable transactions)
cm_take_commit_deallocate_data_permitted	(OSI TP and recoverable transactions)
cm_prepare_ok	recoverable transactions
cm_join_transaction	unchained transactions
sym_dest_name	conversations
sync_level	conversations
cm_none	conversations
cm_confirm	conversations
cm_sync_point	recoverable transactions
cm_sync_point_no_confirm	(OSI TP and recoverable transactions), (recoverable transactions and full-duplex)
timeout	queue-level non-blocking
TP_name	conversations
TP_name_length	conversations
transaction_control	unchained transactions
cm_chained_transactions	unchained transactions
cm_unchained_transactions	unchained transactions
user_field	queue-level non-blocking, callback function
user_field_list	queue-level non-blocking

Appendix H. Solution Developers Program - Enterprise Communications Partners in Development

Program Highlights

To assist your development efforts, IBM offers the Enterprise Communications Partners in Development program.

At no cost to you, you receive:

- Technical assistance from CROSS-PLATFORM specialists in communications, operating systems, and application development, provided for your design or development staff.
- The opportunity to participate in early code programs.
- Opportunities to participate in special marketing activities, such as industry magazine supplements and IBM events.
- Discounts on IBM Communications Server products

Additional fee-based options include:

- Enterprise Communications Remote Test capabilities. Provides members the opportunity to test applications on a variety servers.
- Product education

Membership

The Enterprise Communications Partners in Development program is available to solution developers at no cost.

For additional information or to receive a membership application, Contact:

U.S. and Canada 1-800-553-1623

Worldwide 1-770-835-9902

Fax 1-214-280-6116

email commsrv@vnet.ibm.com

Worldwide Web http://www.austin.ibm.com/developer

© Copyright IBM Corp. 1996, 1998

Glossary

A

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

API. Application programming interface.

application-entity. The part of an application-process that exclusively defines communications formats and protocols for OSI-compliant systems.

application-entity-qualifier. The qualifier that is used to identify a specific instance of an application-entity. It must be unambiguous within the scope of the application-process. See also ISO/IEC 7498-3.

application-process. The part of an open system that performs the information processing for a particular application. See also ISO/IEC 7498:1984.

application-process-title. The unambiguous title of the application-process. It must be unambiguous within the OSI environment. See also ISO/IEC 7498-3.

application context. The set of rules that define the exchange of information between two application programs. See also ISO/IEC 9545.

application context name. The registered name of the application context. See also ISO/IEC 9545.

application programming interface (API). The set of programming language constructs or statements that can be coded in an application program to invoke the specific functions and services provided by an underlying operating system or service program.

association. A relationship between two application-entity instances for the purpose of exchanging data. An association is similar to an SNA LU 6.2 session and is sometimes called a logical connection. See also ISO/IEC 9594.

B

basic conversation. A conversation in which programs exchange data records in an SNA-defined format. This format is a stream of data containing 2-byte length prefixes that specify the amount of data to follow before the next prefix.

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

C

callback function. An application-defined function that is called when an outstanding operation completes.

chained transactions. A series of transactions in which the (n+1)th transaction begins immediately upon the termination of the nth transaction. See also ISO/IEC 10026-1.

Common Programming Interface Communications (CPI-C). A programming interface for applications that require program-to-program communication using the conversational model.

communications resource manager (CRM). The component within a system that manages a particular resource — in this case, a conversational communications resource. See also *X/Open Guide: Distributed Transaction Processing Reference Model*.

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

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

conversation. A logical connection between two programs over an LU type 6.2 session or OSI TP association that allows them to communicate with each other while processing a transaction. See also basic conversation and mapped conversation.

conversation characteristics. The attributes of a conversation that determine the functions and capabilities of programs within the conversation.

conversation partner. One of the two programs involved in a conversation.

conversation queue. A logical grouping of CPI Communications calls on a conversation. Calls associated with a specific queue are processed serially. Calls associated with different queues are processed independently.

conversation state. The condition of a conversation that reflects what the past action on that conversation has been and that determines what the next set of actions may be.

© Copyright IBM Corp. 1996, 1998

CPI-C. Common Programming Interface Communications.

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

D

directory object. A collection of information that is represented in the distributed directory as a single entry. Directory objects consist of attributes, each of which has a type and one or more values.

distinguished name. A completely qualified name that is used to access an entry in a distributed directory. Contrast with *local distinguished name*.

distributed directory. A collection of open systems that cooperate to hold information about **directory objects**. The directory is referred to as distributed because the data can be accessed from multiple locations in a network using local directory interfaces.

DN. See distinguished name.

F

full-duplex. Pertaining to communication in which data can be sent and received at the same time. ***Contrast with half-duplex***.

H

half-duplex. Pertaining to communication in which data can only be transmitted in one direction at a time. ***Contrast with full-duplex***.

I

initialization data. Application-specific data that may be exchanged between two application programs during conversation initialization. See also ISO/IEC 10026-2 for User-Data on TP-BEGIN-DIALOGUE and *SNA Transaction Programmers Reference Manual for LU Type 6.2*.

L

local distinguished name. An incomplete distinguished name for an object in a distributed directory. The complete distinguished name is formed by prefixing the local distinguished name with a system-specific local prefix.

local program. The program being discussed within a particular context. Contrast with remote program.

logical connection. The generic term used to refer to either an SNA LU 6.2 session or an OSI association.

logical unit. A port providing formatting, state synchronization, and other high-level services through which an end user communicates with another end user over an SNA network.

logical unit type 6.2. The SNA logical unit type that supports general communication between programs in a distributed processing environment; the SNA logical unit type on which CPI Communications is built.

M

mapped conversation. A conversation in which programs exchange data records with arbitrary data formats agreed upon by the applications' programmers.

mode name. Part of the CPI Communications side information. The mode name is used by LU 6.2 to designate the properties for the logical connection that will be allocated for a conversation.

N

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

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

O

OSI TP. Refers to the International Standard ISO/IEC 10026, Information Technology — Open Systems Interconnection — Distributed Transaction Processing. ISO/IEC 10026 is one of a set of standards produced to facilitate the interconnection of computer systems.

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

P

partial distinguished name. See local distinguished name.

partner. See conversation partner.

partner principal name. The name by which the distributed security service knows the target partner.

PFID. See program function ID.

principal. Any entity which participates in an authentication exchange.

privilege. An identification that a product or installation defines in order to differentiate SNA service transaction programs from other programs, such as application programs.

program function ID. A unique identifier for the function performed by a particular program. Multiple installations of a program may have equivalent program function IDs (PFID).

protected resource. A local or distributed resource that is updated in a synchronized manner during processing managed by a resource recovery interface and a sync point manager.

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

R

remote program. The program at the other end of a conversation with respect to the reference program. Contrast with local program.

resource recovery interface. An interface to services and facilities that use two-phase commit protocols to coordinate changes to distributed resources.

S

secondary information. Information associated with the return code at the completion of a call. The information can be used to determine the cause of the return code.

session. A logical connection between two logical units that can be activated, tailored to provide various protocols, and deactivated as requested.

side information. System-defined values that are used for the initial values of the *conversation_security_type*, *mode_name*, *partner_LU_name*, *security_password*, *security_user_ID*, and *TP_name* characteristics.

state. See conversation state.

state transition. The act of moving from one conversation state to another.

subordinate program. The application program that issued either Accept_Conversation or Accept_Incoming for a protected conversation.

superior program. The application program that issued Initialize_Conversation for a protected conversation.

symbolic destination name. Variable corresponding to an entry in the side information.

synchronization point. A reference point during transaction processing to which resources can be restored if a failure occurs.

sync point manager. A component of the operating environment that coordinates commit and backout processing among all the protected resources involved in a sync point transaction. Synonymous with transaction manager.

Systems Network Architecture. A description of the logical structure, formats, protocols, and operational sequences for transmitting information units through, and controlling the configuration and operation of, networks.

T

transaction. A related set of operations that are characterized by the ACID (atomicity, consistency, isolation, and durability) properties. See also ISO/IEC 10026-1.

transaction manager. The component within a system that manages the coordination of resources within a transaction. Synonymous with sync point manager. See also *X/Open Guide: Distributed Transaction Processing Reference Model*.

transition. See state transition.

U

unchained transactions. A series of transactions in which the (n+1)th transaction does not begin immediately upon the termination of the nth transaction, but is explicitly started at a later time. See also ISO/IEC 10026-1.

user field. Application data that can be associated with an outstanding operation. The data, as specified by the program, can be returned to the program through a Wait_For_Completion call issued for that outstanding operation, or it can be passed to the callback function associated with the outstanding operation, when the operation completes.

X

X/Open. X/Open is an independent, worldwide, open systems organization whose mission is to bring users greater value from computing, through the practical implementation of open systems.

X/Open TX interface. *X/Open Distributed Transaction Processing: The TX (Transaction Demarcation) Specification* defines an interface between the transaction manager and the application program. It is similar to the IBM SAA resource recovery interface.

Index

Special Characters

- (dash) 112
_ (underscore) 13, 112
. (period) 653
& (ampersand) 112

A

abnormal program ending 23
Accept Conversation (CMACCP) 119
  OS/2 considerations 487, 625
  VM/ESA-specific errors 535
  VM/ESA-specific notes 583
Accept_Incoming (CMACCI) 121
access security information
  on OS/2 486, 625
acknowledgement xxv
advanced program-to-program communications
  See also LU 6.2
  verbs 728
advanced-function calls
  automatic data conversion 42
  examples 74—103
AE_qualifier, defined 23
AIX
  allocation request 391
  compiling 393
  concept 382
  conformance classes 382
  conversation scope 389
  conversation_ID, scope of 389
  dangling conversation, deallocating 388
  deviation 391
  diagnosing error 389
  documentation 381—407
  extension calls 394
  Extract_Conversation_Security_Type (XCECST) 396
  Extract_Conversation_Security_User_ID (XCECSU) 398
  identifying error 389
  languages supported 383
  list 394
  operating environment 382
  product-specific errors 389
  profile, working with 388
  pseudonym file 383
  pseudonym files 383
  reference publications 381
  security, using 391
  Set_Conversation_Security_Password (XCSCSP) 399
AIX *(continued)*
  Set_Conversation_Security_Type (XCSCST) 401
  Set_Conversation_Security_User_ID (XCSCSU) 403
  side information 383
  starting SMIT 388
  transaction program, running 393
alias name for partner LU on OS/2 493, 496, 630, 632
Allocate (CMALLC) 124
  OS/2 considerations
  VM/ESA-specific errors 535
  VM/ESA-specific notes 583
allocate_confirm 282
allocating data buffers on OS/2 500
allocation requests, when sent
  MVS/ESA 424
  OS/2 450
  OS/400 524
  VM/ESA 540
allocation wrapback in VM/ESA 539, 582
AP_title 284
AP_title, defined 23
APPC
  See advanced program-to-program communications
APPC/MVS services, use with CPI Communications 425
APPCLLU OS/2 environment variable 597, 499, 491, 498, 629
APPCTPN OS/2 environment variable 597, 499
Application Generator considerations
  general
  in OS/400 515
  in VM/ESA 529
application migration 735
  from X/Open CPI-C to CPI-C 2.0 735
application TP names on OS/2 496, 632
application_context_name 286
application_context_name, defined 23
ASCII, conversion of 652
Assembler considerations (VM/ESA) 529
attach manager-started programs on OS/2 499

B

backout-required condition
  described 58
  effect on multiple conversations 52
basic conversation 19, 256
begin conversation
  CMALLC (Allocate) 124
  example flow 69, 84, 88
  program startup 22

© Copyright IBM Corp. 1996, 1998

begin conversation *(continued)*
  simple example 29
begin_transaction 288
blank *sym_dest_name* 25, 200
blocking operations 47
buffering of data
  description 44, 256
  example flow 70

C

C considerations
  general 112
  in Networking Services for Windows 431
    function calls 431
    pseudonym files 431
  in OS/2 438
  in VM/ESA 529
call reference
  format 107
  how to use 113
  in VM/ESA 528
  parameters 107
  related information 107
  RPG 113
  state changes 107
  usage notes 107
call_ID 642
callback_function 337
callback_info 337
calls
  advanced-function
    examples 74—103
  call characteristics
    See characteristics, call characteristics
  for resource recovery interfaces 54
  format for AIX 394—407
  format for MVS/ESA 419
  format for Networking Services for Windows 430
  format for OS/2 454, 601
    conversation, overview 471, 610
  format for VM/ESA 528
  naming convention 13
  possible values, table 642
  starter-set
    examples 68—73
Cancel_Conversation (CMCANC) 131
chained transactions 61, 193
changing data flow direction
  by receiving program 75
  by sending program 72, 74
changing side information on OS/2 443
character sets
  exceptions for SNA TP names 727
  general 647
  on OS/2 496, 632
character sets *(continued)*
  T61String 647
character strings 649
characteristics
  automatic conversion of 41
  call characteristics
    AE_qualifier 280
    allocate_confirm 282
    AP_title 284
    application_context_name 286
    begin_transaction 288
    call_ID 642
    confirmation_urgency 290, 642
    context_ID 161
    conversation_queue 642
    conversation_return_code 642
    conversation_security_type 295
    conversation_state 163
    conversation_type 166
    deallocate_type 303
    directory_encoding 642
    directory_syntax 642
    error_direction 307
    expedited_receive_type 642
    fill 310
    initialization_data 312
    log_data 316
    mode_name 175
    OOID_list_count 642
    partner_ID_scope 642
    partner_ID_type 642
    partner_LU_name 180
    prepare_data_permitted 329
    prepare_to_receive_type 331
    processing_mode 334
    queue_callback_function 337
    queue_processing_mode 340, 642
    receive_type 344
    return_control 346
    security_password 292
    security_user_ID 185
    send_receive_mode 349
    send_type 351
    sync_level 354
    TP_name 357
    transaction_control 359, 642
  comparison of defaults 35
  default values 30
  examining 34
  initial values
    on OS/2, set by Accept_Conversation 487, 626
    on OS/2, set by Initialize_Conversation 491, 628
  initial values, table 35
  integer values 641
  modifying 34
  naming convention 13

characteristics *(continued)*
OS/2 additions 443
pseudonym 14
viewing 34
CICS/ESA
allocation requests 412
conformance classes 408
documentation 407—414
extension calls 413
reference publications 407
CMACCI (Accept_Incoming) 121
CMACCP (Accept_Conversation) 119
OS/2 considerations 487, 625
VM/ESA-specific errors 535
VM/ESA-specific notes 583
CMALLC (Allocate) 124
OS/2 considerations
VM/ESA-specific errors 535
VM/ESA-specific notes 583
CMC COPY (VM/ESA) 530
CMCANC (Cancel_Conversation) 131
CMCFM (Confirm) 133
protocol error in VM/ESA 540
VM/ESA-specific errors 535
CMCFMD (Confirmed) 137
CMCNVI (Convert_Incoming) 139
CMCNVO (Convert_Outgoing) 141
CMCSP COPY (VM/ESA) 530
CMD.EXE program for OS/2 440
CMDEAL (Deallocate) 143
Networking Services for Windows considerations 433
protocol error in VM/ESA 540
VM/ESA-specific errors 535, 536
CMDFDE (Deferred_Deallocate) 153
CMEACN (Extract_Application_Context_Name) 159
CMEAEQ (Extract_AE_Qualifier) 155
CMEAPT (Extract_AP_Title) 157
CMECS (Extract_Conversation_State) 163
CMECT (Extract_Conversation_Type) 166
CMECTX (Extract_Conversation_Context) 161
CMEID (Extract_Initialization_Data) 168
CMEMBS (Extract_Maximum_Buffer_Size) 173
CMEMID (Extract_Mapped_Initialization_Data) 170
CMEMN (Extract_Mode_Name) 175
CMEPID (Extract_Partner_ID) 177
CMEPLN (Extract_Partner_LU_Name) 180
CMESI (Extract_Secondary_Information) 182
CMESL (Extract_Sync_Level) 189
CMESRM (Extract_Send_Receive_Mode) 187
CMESUI (Extract_Security_User_ID) 185
AIX call 398
OS/2 call 473, 612
CMETC (Extract_Transaction_Control) 193
CMETPN (Extract_TP_Name) 191
CMFLUS (Flush) 195
VM/ESA-specific errors 536, 584
CMFORTRN COPY (VM/ESA) 530
CMHASM COPY (VM/ESA) 530
CMINCL (Include_Partner_IN_Transaction) 198
CMINIC (Initialize_For_Incoming) 203
CMINIT (Initialize_Conversation) 200
OS/2 considerations 490, 628
VM/ESA-specific errors 536, 584
CMPASCAL COPY (VM/ESA) 530
CMPREP (Prepare) 205
CMPTR (Prepare_To_Receive) 208
protocol error in VM/ESA 540
VM/ESA-specific errors 536
CMRCV (Receive) 213
OS/2 considerations 492, 629
protocol error in VM/ESA 540
VM/ESA-specific errors 536
CMRCVM (Receive_Mapped_Data) 231
CMRCVX (Receive_Expedited_Data) 228
CMREXX COPY (VM/ESA) 530
CMRLTP (Release_Local_TP_Name) 244
CMRTS (Request_To_Send) 246
CMSAC (Set_Allocate_Confirm) 282
CMSACN (Set Application_Context_Name) 286
CMSAEQ (Set_AE_Qualifier) 280
CMSAPT (Set_AP_Title) 284
CMSBT (Set_Begin_Transaction) 288
CMSCT (Set_Conversation_Type) 301
CMSCU (Set_Confirmation_Urgency) 290
CMSDT (Set_Deallocate_Type) 303
CMSED (Set_Error_Direction) 307
CMSEND (Send_Data) 249
OS/2 considerations 492, 630
protocol error in VM/ESA 540
VM/ESA-specific errors 536, 585
VM/ESA-specific notes 537
CMSERR (Send_Error) 259
protocol error in VM/ESA 540
VM/ESA-specific errors 536
CMSF (Set_Fill) 310
CMSID (Set_Initialization_Data) 312
CMSJT (Set_Join_Transaction) 314
CMSLD (Set_Log_Data) 316
Networking Services for Windows considerations 433
OS/2 considerations 448
VM/ESA-specific errors 536, 585
CMSLTP (Specify_Local_TP_Name) 361
CMSMID (Set_Mapped_Initialization_Data 318
CMSMN (Set_Mode_Name) 321
Networking Services for Windows considerations 432
OS/2 considerations
CMSNDM (Send_Mapped_Data) 271

CMSNDX (Send_Expedited_Data) 268
CMSPDP (Set_Prepare_Data_Permitted) 329
CMSPID (Set_Partner_ID) 323
CMSPLN (Set_Partner_LU_Name) 327
  OS/2 considerations 493, 630
  VM/ESA-specific errors 536
CMSPM (Set_Processing_Mode) 334
CMSPTR (Set_Prepare_To_Receive_Type) 331
CMSQCF (Set_Queue_Callback_Function) 337
CMSQPM (Set_Queue_Processing_Mode) 340
CMSRC (Set_Return_Control) 346
CMSRT (Set_Receive_Type) 344
CMSSAA TXTLIB (VM/ESA) 529
CMSSL (Set_Sync_Level) 354
  OS/2 considerations 451
CMSSRM (Set_Send_Receive_Mode) 349
CMSST (Set_Send_Type) 351
CMSTC (Set_Transaction_Control) 359
CMSTPN (Set_TP_Name) 357
  OS/2 considerations
CMTRTS (Test_Request_To_Send_Received) 363
CMWAIT (Wait_For_Conversation) 369
CMWCMP (Wait_For_Completion) 366
CNSI (CPI Communications name service interface) 739
  attributes
    complex 740
    simple 740
    single-valued 740
  encoding for complex attributes
    fields 742
  profile object 740
  program_installation object 741
  sample calls 739
  scenarios 742
  server object 740
COBOL considerations 112
  in OS/2 439
  in VM/ESA 528
commit call
  by sending program 98
  conversation deallocation before 102
  example flow 98, 101, 103
  with conversation state change 100
commit tree, illustrated 63
common programming interface (CPI)
  communications
    See CPI Communications
communication
  across an SNA network 18
  resource manager (CRM) 18
  with an APPC program 727
communications directory, CMS (VM/ESA) 532
Communications Manager
  See OS/2
completed_op_count 366
completed_op_index_list 366
concurrent conversations 30
concurrent operations
  conversation queues 44
  multiple program threads 44
    association of calls, table 45
Confirm (CMCFM) 133
  protocol error in VM/ESA 540
  VM/ESA-specific errors 535
Confirm state 52
Confirm-Deallocate state 52
Confirm-Send state 52
confirmation processing
  Confirm call 133
  Confirmed call 137
  example flow 79
confirmation_urgency 642
Confirmed (CMCFMD) 137
conformance class 745
  callback function 749
  conversation-level non-blocking 749
  conversations 746
  data conversion routines 750
  directory 751
  distributed security 751
  expedited data 751
  full-duplex 751
  LU 6.2 747
  OSI TP 747
  product implementation, table 109
  queue-level non-blocking 749
  recoverable transactions 748
  secondary information 752
  security 750
  server 750
  unchained transactions 748
contexts
  changing 32
  creating 32
  current 32
  identifier 32
  relationship
    conversation 32
    security parameter 32
  setting 32
control_information_received 642
control_information_received parameter 29
convention, naming 13
conversation
  accept 119
  allocate 124
  basic 19, 256
  canceling 131
  characteristics
    See also characteristics, call characteristics
    described 33

conversation *(continued)*
characteristics *(continued)*
overview 33
concurrent 30
dangling
See dangling conversation, deallocating
deallocate 143
description 19
effects on 724
examples 29, 68—103
full-duplex
setting up 86
terminating 89
using 86
identifier 29
illustration
inbound and outbound 32
initialize 200
mapped 19, 256
multiple 30
multiple inbound
illustration 30, 31
in server programs 30
multiple outbound
conversation_ID 22
illustration 31
queues 90
security 51
states
See states, conversation
synchronization and control
Confirm call 133
Confirmed call 137
Flush call 195
Prepare_To_Receive call 208
Request_To_Send call 246
Send_Error 259
Test_Request_To_Send_Received 363
transition from a state 52
types 19
conversation calls on OS/2 471, 610
conversation_ID
described 29
scope of
in CICS/ESA 412
in MVS/ESA 420
in OS/2 447
in OS/400 519
in VM/ESA 534
sharing across tasks, in MVS/ESA 420
conversation security
access security information 51
incompatible values, table 52
on OS/2 486, 625
conversation_security_type characteristic
on OS/2 496, 632
conversation_security_type characteristic *(continued)*
on VM/ESA 564, 579
conversation states
additional CPI states 64
description 52
extracting 163
full-duplex protected conversations (CPIRR) 723
full-duplex protected conversations (X/Open) 724
half-duplex protected conversations (CPIRR) 710
half-duplex protected conversations (X/Open) 695, 711
list 52
possible values 642
pseudonym 13
table
full-duplex 718
half-duplex 704
valid for resource recovery 65
conversation type characteristic 301
conversation_queue 337, 642
conversation_return_code 642
conversation_security_type, defined 24
conversion
characteristics 41
data 43
conversion of characteristics
on OS/2 495
conversion to and from ASCII
on OS/2 495
Convert_Incoming (CMCNVI) 139
Convert_Outgoing (CMCNVO) 141
Coordinated Resource Recovery (VM/ESA) 579
CPI Communications
additional information sources 8
audience 5
communication with APPC programs 727
conversational model 5
CPI-C 1.0 10
CPI-C 1.1 10
CPI-C 1.2 10
CPI-C 2.0 11
directory object 20
functional levels 10
in SNA networks 18
introducing 5
name service interface (CNSI) 738
naming convention 13
program operating environment 21
related APPC/LU 6.2 publications 8
relationship to 2.1 spec 6
relationship to LU 6.2 interface 725
relationship to products 7
versions of, table 12
what's new in this release 6
with resource recovery interfaces 54
X/Open CPI-C 2.0 11

CPI Communications *(continued)*
X/Open extensions 10
CPI-C
See CPI Communications
CPIC.LIB for OS/2 link edit 439
CPICOBOL.LIB for OS/2 link edit 439
CPICOMM LOGDATA file on VM/ESA 534, 539
CPICOMM REXX environment for OS/2 440
CPICREXX.EXE program for OS/2 440
CPIFORTN.LIB for OS/2 link edit 439
CPSVCMG mode name
OS/2 considerations
CRM
characteristic values 39
for LU 6.2 18, 39
for OSI TP 39, 692
using particular type 39
CSP considerations 112

D

dangling conversation, deallocating 22
in AIX 388
in CICS/ESA 411
in MVS/ESA 420
in Networking Services for Windows 433
in OS/2 447
in OS/400 518
in VM/ESA 534
data
buffer allocation on OS/2 500
direction, changing
by receiving program 75
by sending program 72, 74
flow
in both directions 72
in one direction 69, 84, 88
purging 80, 265
reception and validation of 78
data records
description 19
Receive call 214
Send_Data call 256
data_received parameter 29
DCE directory
DCE interface 737
directory objects 738
interaction of 738
extensions 737
profile object 737
server group object 738
server object 737
Deallocate (CMDEAL) 143
Networking Services for Windows considerations 433
protocol error in VM/ESA 540
Deallocate (CMDEAL) *(continued)*
VM/ESA-specific errors 535, 536
deallocate_type characteristic 143
defaults for TPs on OS/2 498
Deferred_Deallocate (CMDFDE) 153
defining a program (TP) on OS/2 498
defining side information on OS/2 443
Delete_CPIC_Side_Information (XCMDSI)
OS/2 call 455, 602
destination name, symbolic
blank 25, 200
defined 21
example 69, 84, 88
OS/2 considerations 496, 632
deviations from CPI Communications architecture
CICS/ESA 413
MVS/ESA 424
OS/2 450
VM/ESA 540
diagnosing errors
CICS/ESA 412
MVS/ESA 423
Networking Services for Windows 433
OS/2 448
OS/400 520
VM/ESA 536
directory_encoding 323, 642
directory_syntax 323, 642
distinguished name, naming convention 655
distinguished_name, defined 24
distributed directory
defined 25
directory object 25
distinguished name (DN) 23
illustration of
generic program interaction 25
interaction with CPI Communications 27
locating partner program 25, 96
program function identifier (PFID) 25
program interaction 25
using 25
distributed security
illustration of CRM interaction 28
principal names 28
DN, specifying default 323
double-byte TP names on OS/2 452, 497, 633

E

EBCDIC, conversion to 139, 750
on OS/2 495
environment requirements for MVS/ESA 418
environment variables on OS/2
for local LU name (APPCLLU) 597, 499, 491, 498, 629
for TP name (APPCTPN) 597, 499

environment, operating
CICS/ESA 407
MVS/ESA 418
OS/2 437
OS/400 513
VM/ESA 528
error codes (VM/ESA) 528
error_direction characteristic
and Send-Pending state 82, 726
error reporting
example 80
OS/2 considerations with REXX 441
Send_Error call 264
errors, diagnosing
CICS/ESA 412
MVS/ESA 423
Networking Services for Windows 433
OS/2 448
OS/400 520
VM/ESA 536
errors, product-specific
See product-specific errors, identifying
event (VM/ESA)
communications 585
services 585
system 585
event management in VM/ESA
using for CPI Communications 569, 575, 585
event management services (VM/ESA) 569, 575, 585
examining conversation characteristics 34
See also extract calls
example flows
See tutorial information
expedited_receive_type 642
extension calls, CPI Communications
CICS/ESA 413
MVS/ESA 425
OS/2 454—485, 601—624
OS/400 524
VM/ESA 541—576
extract calls
AE Qualifier (CMEAEQ) 155
AP_Title (CMEAPT) 157
Application_Context_Name (CMEACN) 159
Conversation_Context (CMECTX) 161
Conversation_Security_Type (XCECST) 472, 611
on OS/2 472, 611
Conversation_Security_User_ID (XCECSU)
on OS/2
Conversation_State (CMECS) 163
Conversation_Type (CMECT) 166
Initialization_Data (CMEID) 168
Mapped_Initialization_Data (CMEMID) 170
Maximum_Buffer_Size (CMEMBS) 173
Mode_Name (CMEMN) 175
Partner_ID (CMEPID) 177
extract calls *(continued)*
Partner_LU_Name (CMEPLN) 180
Secondary_Information (CMESI) 182
Security_User_ID (CMESUI) 185
Send_Receive_Mode (CMESRM) 187
Side_Info_Entry (XCMESI) (OS/2) 457, 604
Sync_Level (CMESL) 189
TP_Name (CMETPN) 191
Transaction_Control (CMETC) 193
Extract_AE_Qualifier (CMEAEQ) 155
Extract_AP_Title (CMEAPT) 157
Extract_Application_Context_Name (CMEACN) 159
Extract_Conversation_Context (CMECTX) 161
Extract_Conversation_LUWID (XCECL) call on VM/ESA 544
Extract_Conversation_Security_Type (XCECST)
AIX call 396
OS/2 call 472, 611
Extract_Conversation_State (CMECS) 163
Extract_Conversation_Type (CMECT) 166
Extract_Conversation_Workunit_ID (XCECWU) call on VM/ESA 548
Extract_CPIC_Side_Information (XCMESI)
OS/2 call 457, 604
Extract_Initialization_Data (CMEID) 168
Extract_Local_Fully_Qualified_LU_Name (XCELFQ)
VM/ESA call 550
Extract_Mapped_Initialization_Data (CMEMID) 170
Extract_Maximum_Buffer_Size (CMEMBS) 173
Extract_Mode_Name (CMEMN) 175
Extract_Partner_ID (CMEPID) 177
Extract_Partner_LU_Name (CMEPLN) 180
Extract_Remote_Fully_Qualified_LU_Name (XCERFQ)
VM/ESA call 552
Extract_Secondary_Information (CMESI) 182
Extract_Security_User_ID (CMESUI) 185
AIX call 398
OS/2 call 473, 612
Extract_Send_Receive_Mode (CMESRM) 187
Extract_Sync_Level (CMESL) 189
Extract_TP_Name (CMETPN) 191
Extract_TP_Name (XCETPN) call on VM/ESA 554
Extract_Transaction_Control (CMETC) 193

F

fields
defined for OS/2 494, 631
of side_info_entry on OS/2 458, 460, 607
fill characteristic 310
flow
definition of 67
diagrams 69—103
Flush (CMFLUS) 195
VM/ESA-specific errors 536, 584

FORTRAN considerations
  general 112
  in OS/2 439
  in VM/ESA 529
full-duplex
  See states, conversation

G

graphic representations for character sets
  OS/2 additions 495
  table 647
green ink, use of 7, 107

H

half-duplex
  See states, conversation
HELP, VM/ESA online 588

I

identifier (PFID) 25
IMS/ESA
  documentation 415
  reference publications 415
Include_Partner_In_Transaction (CMINCL)
  subordinate 198
initialize
  conversation 200
  state 52
Initialize_Conv_For_TP (XCINCT)
  OS/2 call 474, 613
Initialize_Conversation (CMINIT) 200
  OS/2 considerations 490, 628
  VM/ESA-specific errors 536, 584
Initialize_For_Incoming (CMINIC) 203
Initialize_Incoming state 53
integer
  values 641
integer values
  OS/2 additions
interface, communications
  See CPI Communications
invoking routines
  in MVS/ESA 419
  in VM/ESA 528

K

key variable on OS/2 496, 632
keylock feature on OS/2 445
keys in VM/ESA 586

L

language considerations, programming 111
  in MVS/ESA 418
  in OS/2 437
  in OS/400 515
  in VM/ESA 529
languages supported
  CICS/ESA 408
  MVS/ESA 419
  OS/2 437
  OS/400 515
  VM/ESA 528
licensing agreement xxiii
link edit for OS/2 439
linkage conventions in MVS/ESA 419
load module for CPI Communications in MVS/ESA 418
local partner 20
locating partner program
  using distributed directory 96
log_data characteristic
  OS/2 considerations 448
  set 316
logical connection
  association 18
  session 18
logical records
  description 19
  OS/2 considerations 492, 629
  Receive call 214
  Send_Data call 256
logical unit
  See also LU 6.2
  illustration 18
logical unit of work identifier (LUWID) format on VM/ESA 545
logical unit of work identifier on OS/2 486
LU
  See logical unit
LU 6.2
  and CPI communications calls 725
    relationship of verbs, table 728—734
  considerations in VM/ESA 580
  related information 725—728
luname tag in VM/ESA 533
LUWID (logical unit of work identifier) format on VM/ESA 545

M

MAP_NAME 725
mapped conversation 19, 256
mapping
  &I2@CRMMAP.
    CPI-C to OSI TP services
    LU 6.2 and OSI TP CRMs

mapping *(continued)*
&I2@CRMMAP. *(continued)*
OSI TP services to CPI-C half-duplex conversations
CRMs
full-duplex conversations
OSI
TP
half-duplex conversations
match keys in VM/ESA 586
mode name, defined 24
mode_name characteristic
defined 24
extract 175
length 652
OS/2 considerations 496, 632
set 321
mode_name CPSVCMG
OS/2 considerations 653
mode_name SNASVCMG
Allocate call 125
OS/2 considerations 653
Set_Mode_Name call 321
mode, processing 334
modename tag in VM/ESA 533
modifying conversation characteristics 34
See also set calls
multiple conversations
inbound 30
outbound 30
multiple program threads 45
multitasking, CMS (VM/ESA)
using event management services 575, 585
MVS/ESA
conformance classes 418
documentation 417—427
errors 421
reference publications 417

N

naming conventions 13
native encoding on OS/2 (ASCII) 494
network name for partner LU
OS/2 considerations 493, 496, 630, 632
Networking Services for Windows
conformance classes 429
dangling conversations in 433
deviations from CPI-C 434
diagnosing errors 433
documentation 429—434
linking with CPI-C import library 432
memory considerations 432
data buffers 432
stack size 432
mode_name 432
Networking Services for Windows *(continued)*
product-specific errors 433
pseudonym files 430—431
reference publications 429
side information 432
nick tag in VM/ESA 533
node services 22
non-blocking operations
calls returning incomplete, table 48
context management 50
conversation-level 48
operations 47
outstanding operation 47
processing_mode 48
queue-level
callback function 49
using 49
wait facility 49
non-queued programs on OS/2 499

O

online HELP Facility (VM/ESA) 588
OOID_list 366
OOID_list_count 366, 642
operating environment
CICS/ESA 407
example in CPI 21
generic elements 21
illustration 21
MVS/ESA 418
OS/2 437
OS/400 513
VM/ESA 528
Operating System/2
See also OS/2
documentation 435—511
Operating System/400
See also OS/400
documentation 513—526
OS/2 435
characteristics, fields, and variables 494, 631
conformance classes 437
considerations for CPI Communications calls 485, 624
conversation calls 471—478, 610—617
CPI Communications functions not available 452
defining and running a program 498
extension calls
Delete_CPIC_Side_Information (XCMDSI) 455, 602
Extract_Conversation_Security_Type (XCECST) 472, 611
Extract_Conversation_Security_User_ID (XCECSU) 473, 612
Extract_CPIC_Side_Information (XCMESI) 457, 604

OS/2 *(continued)*
extension calls *(continued)*
Initialize_Conv_For_TP (XCINCT) 474, 613
Set_Conversation_Security_Password (XCSCSP) 476, 615
Set_Conversation_Security_Type (XCSCST) 477, 616
Set_Conversation_Security_User_ID (XCSCSU) 478, 617
Set_CPIC_Side_Information (XCMSSI) 460, 607
programming languages supported 437
pseudonym files
reference publications 436
sample programs 502
side information 443
system management calls 454—471, 601
transaction program control 479—485, 618—624
OS/400
conformance classes 515
considerations, programming language
See OS/400, programming language considerations
conversation_ID, scope of 519
conversation support, multiple 525
incoming conversations, prestarting jobs 524
jobs 513
log_data 521
multiple conversation support 525
node services
dangling conversations, ending 518
reclaim resource processing 519
operating environment
Application Generator 515
C/400 515
communications side information, described 516
Cross System Product (CSP) 515
CSP/Application Development 515
CSP/Application Execution 515
FORTRAN/400 516
REXX 516
REXX error and failure conditions 523
REXX reserved RC variable 522
side information, communications 516
overview 513
portability considerations 525
prestarting jobs for incoming conversations 524
programming language considerations 515
pseudonym files 516
reference publications 513
return codes 519, 521
reasons for an error return_code, determining 520
scope of a *conversation_ID* 519
side information, communications
described 516
subsystems 514
OS/400 *(continued)*
support of *log_data* 521
terms and concepts 513
converation support, multiple 525
conversation_ID, scope of 519
incoming conversations, prestarting jobs 524
jobs 513
multiple converation support 525
prestarting jobs for incoming conversations 524
scope of a *conversation_ID* 519
subsystems 514
outstanding operations 48, 369

P

parameters
input 29
output 29
partner
identify partner program
distributed directory 20
program supplied 20
side information 20
install
naming programs 30
program binding 22, 25
program function ID 20, 25
program installation object 25
partner_ID 323
partner_ID_scope 323, 642
partner_ID_type 323, 642
partner_LU_name characteristic
extract 180
length 652
OS/2 considerations 496, 632
set 327
partner_LU_name, defined 23
Pascal considerations (VM/ESA) 529
password tag in VM/ESA 533
performance considerations
for CPI Communications calls in MVS/ESA 425
PFID (program function identifier)
defined 656
relationship to DNs, illustrated 656
PFID assignment algorithms 656
PIP data 725
PL/I considerations 112
Prepare (CMPREP)
preparing for commit 205
prepare_data_permitted 642
Prepare_To_Receive (CMPTR) 208
protocol error in VM/ESA 540
VM/ESA-specific errors 536
prepare_to_receive_type characteristic 208
Procedures Language (REXX) considerations
general

Procedures Language (REXX) considerations *(continued)*
in OS/2 440
in VM/ESA 530
processing requirements for MVS/ESA 418
product publications
AIX publications 381
CICS/ESA publications 407
IMS/ESA publications 415
MVS/ESA publications 417
Networking Services for Windows publications 429
OS/2 publications 436
OS/400 publications 513
VM/ESA publications 527
product-specific errors, identifying
CICS/ESA 412
MVS/ESA 421
OS/2 447
OS/400 519
VM/ESA 534
program
asynchronous updates of variables 112
calls 29
compilation 28
partners 20
startup processing 22
states
See states, conversation
termination processing 22
TP definition on OS/2 498
types on OS/2
attach manager-started 499
non-queued 499
program binding
defined 657
errors 658
extracting 658
fields
table 658
sample 658
program defined side information on OS/2 444
programming language considerations 111
in MVS/ESA 418
in OS/2 437
in OS/400 515
in VM/ESA 529
protected conversation 54
protected resource 54
protocol errors (VM/ESA) 539
pseudonym
example 14
explanation 13
values 641
OS/2 additions 443
pseudonym files
for CICS/ESA 408
pseudonym files *(continued)*
for MVS/ESA 419
for OS/2 443
for OS/400 516
for VM/ESA 530

Q

queue-level
example flow 91
non-blocking 90
queue_processing_mode 642
queues
calls associated with 44
conversation queues 44

R

Receive (CMRCV) 213
OS/2 considerations 492, 629
protocol error in VM/ESA 540
VM/ESA-specific errors 536
Receive state
description 52
how a program enters 247
Receive_Expedited_Data (CMRCVX)
expedited_receive_type 228
Receive_Mapped_Data (CMRCVM) 231
receive_type characteristic 344
records, logical
description 19
OS/2 considerations 492, 629
Receive call 214
Send_Data call 256
relative distinguished_name, defined 655
Release_Local_TP_Name (CMRLTP) 244
remote partner
definition 20
residing on local system 20
reporting errors
example 80
OS/2 considerations with REXX 441
Send_Error call 264
Request_To_Send (CMRTS) 246
Reset state 52
resource recovery interfaces
described 54
VM/ESA support 579
return codes 642, 661—678
return_code parameter
definitions of values 661—678
described 29
possible values 642
return_control characteristic 346
REXX considerations
general

REXX considerations *(continued)*
  in OS/2 440
  in VM/ESA 530

S

SAA resource recovery interface
  VM/ESA support 579
  with CPI Communications 54
sample programs
  CPI Communications (generic)
  for OS/2 502
secondary information
  application-oriented 679
  CPI-defined 679
    table 681
  CRM-specific
    examples from LU 6.2 692
  different types and condition codes 680
    table 680
  implementation-related
    examples 693
  types and return codes
    not associated with 679
    table 679
security information
  on OS/2 486, 625
security tag in VM/ESA 533
security_password characteristic
  on OS/2 496, 632
security_password_length characteristic
  on OS/2 496, 632
security_password, defined 292
security_type characteristic
  on VM/ESA 564, 579
security_user_ID characteristic
  on OS/2 496, 632
security_user_ID_length characteristic
  on OS/2 496, 632
security_user_ID, defined 24
Send state
  description 52
Send-Pending state
  and *error_direction* characteristic 726
  description 52
  error direction 82
send-receive mode
  characteristic values
    not set for full-duplex, table 19, 40
    not set for half-duplex, table 19, 40
  send control 19
Send_Data (CMSEND) 249
  OS/2 considerations 492, 630
  protocol error in VM/ESA 540
  VM/ESA-specific errors 536, 585
  VM/ESA-specific notes 537
Send_Error (CMSERR) 259
  protocol error in VM/ESA 540
  VM/ESA-specific errors 536
Send_Expedited_Data (CMSNDX) 268
Send_Mapped_Data (CMSNDM) 271
send_type characteristic 351
service transaction programs
  general 727
  OS/2 considerations
set calls
  AE_qualifier 280
  allocate_confirm 282
  AP_title 284
  application_context_name 286
  begin_transaction 288
  confirmation_urgency 290
  conversation_security_password 292
    on OS/2
  conversation_security_type 295
    on OS/2
  conversation_security_user 298
    on OS/2
  conversation_type 301
  deallocate_type 303
  error_direction 307
  fill 310
  initialization_data 312
  join_transaction 314
  log_data 316
  mode_name 321
  partner_ID 323
  partner_LU_name 327
  prepare_data_permitted 329
  prepare_to_receive_type 331
  processing_mode 334
  queue_callback_function 337
  queue_processing_mode 340
  receive_type 344
  return_control 346
  send_receive_mode 349
  send_type 351
  side_info_entry (on OS/2) 460, 607
  sync_level 354
  TP_name 357
  transaction_control 359
SET COMDIR command (CMS) 533
Set_AE_Qualifier (CMSAEQ) 280
Set_Allocate_Confirm (CMSAC) 282
Set_AP_Title (CMSAPT) 284
Set_Application_Context_Name (CMSACN) 286
Set_Begin_Transaction (CMSBT) 288
Set_Confirmation_Urgency (CMSCU) 290
Set_Conversation_Security_Password (XCSCSP)
  AIX call 399
  OS/2 call 476, 615

Set_Conversation_Security_Type (XCSCST)
  AIX call 401
  OS/2 call 477, 616
Set_Conversation_Security_User_ID (XCSCSU)
  AIX call 403
  OS/2 call 478, 617
Set_Conversation_Type (CMSCT) 301
Set_CPIC_Side_Information (XCMSSI)
  OS/2 call 460, 607
Set_Deallocate_Type (CMSDT) 303
Set_Error_Direction (CMSED) 307
Set_Fill (CMSF) 310
Set_Initialization_Data (CMSID) 312
Set_Join_Transaction (CMSJT) 314
Set_Log_Data (CMSLD) 316
  Networking Services for Windows considerations 433
  OS/2 considerations 448
  VM/ESA-specific errors 536, 585
Set_Mapped_Initialization_Data (CMSMID) 318
Set_Mode_Name (CMSMN) 321
  Networking Services for Windows considerations 432
  OS/2 considerations
Set_Partner_ID (CMSPID) 323
Set_Partner_LU_Name (CMSPLN) 327
  OS/2 considerations 493, 630
  VM/ESA-specific errors 536
Set_Prepare_Data_Permitted (CMSPDP) 329
Set_Prepare_To_Receive_Type (CMSPTR) 331
Set_Processing_Mode (CMSPM) 334
Set_Queue_Callback_Function (CMSQCF) 337
Set_Queue_Processing_Mode (CMSQPM) 340
Set_Receive_Type (CMSRT) 344
Set_Return_Control (CMSRC) 346
Set_Send_Receive_Mode (CMSSRM) 349
Set_Send_Type (CMSST) 351
Set_Sync_Level (CMSSL) 354
  OS/2 considerations 451
Set_TP_Name (CMSTPN) 357
  OS/2 considerations
Set_Transaction_Control (CMSTC) 359
shared memory, allocation of data buffers on OS/2 500
side information, defining
  in CICS/ESA 409
  in MVS/ESA 420
  in OS/2 443
  in OS/400 516
  in VM/ESA 532
  overview 22
  parameters, on OS/2
    conversation_security_type 446
    mode_name 445
    partner_LU_name 445
    security_password 446
    security_user_ID 446
side information, defining *(continued)*
  parameters, on OS/2 *(continued)*
    sym_dest_name 445
    TP_name 445
    TP_name_type 445
  purpose 23
  setting and accessing 22
side_info_entry structure layout for OS/2 calls
  Extract_CPIC_Side_Information (XCMESI) 458
  Set_CPIC_Side_Information (XCMSSI) 460, 607
side_info_entry variable on OS/2 496, 632
side_info_entry_length variable on OS/2 496, 632
Signal_User_Event (XCSUE) call on VM/ESA 568
SNA
  network 18
  service transaction programs 727
SNA service TP names on OS/2 452, 496, 632
SNASVCMG mode name
  OS/2 considerations
special notes for CPI Communications products
  CICS/ESA 414
  MVS/ESA 425
  OS/2 485, 624
  OS/400 524
  VM/ESA 577
Specify_Local_TP_Name (CMSLTP) 361
starter-set calls
  examples 68—73
state tables, full-duplex
  abbreviations
    conversation characteristics 712
    conversation queues 713
    data_received and status_received 715
    return code values 713
  table symbols 716
state tables, half-duplex 724
  abbreviations 697
    conversation characteristics 697
    conversation queues 699
    data_received and status_received 702
    return code values 700
  example of how to use 695
  table symbols 703
state transition 52
states, conversation
  additional CPI states 64
  description 52
  extracting 163
  full-duplex protected conversations (CPIRR) 723
  full-duplex protected conversations (X/Open) 724
  half-duplex protected conversations (CPIRR) 710
  half-duplex protected conversations (X/Open) 695, 711
  list 52
  possible values 642
  pseudonym 13

states, conversation *(continued)*
table
full-duplex 718
half-duplex 704
valid for resource recovery 65
status_received parameter 29
strings, character 649
subordinate program 63
superior program 63
sym_dest_name
See symbolic destination name
symbolic destination name
blank 25, 200
defined 21
example 69, 84, 88
OS/2 considerations 496, 632
sync point
described 55
logical unit of work 55
sync point manager (SPM) 55
transaction manager 55
sync_level characteristic 354
OS/2 considerations 451
synchronization point
See sync point
Systems Network Architecture
network 18
service transaction programs 727

T

T61String 647
take-backout notification
responses to
table 59
take-commit notification
responses to
table 59
Test_Request_To_Send_Received (CMTRTS) 363
TP definition on OS/2 498
TP names not supported on OS/2 452
TP profiles
in MVS/ESA 425
TP-model application in VM/ESA 580
TP_name characteristic
extract 191, 554
Networking Services for Windows considerations 433
OS/2 considerations 496, 632
set 357
TP_name_type field on OS/2 496, 632
TP_name, defined 23
tpn tag in VM/ESA 533
transaction
chained 61
join
explicit request 61
transaction *(continued)*
join *(continued)*
implicit request 61
responses, table
unchained 61
transaction_control, chained or unchained 359
transition, state 52
tutorial information
example flows 67—103
terms and concepts 17—66
types of conversations 19

U

unchained transactions 61
user profile managment on OS/2
user-defined side information on OS/2 444
USER_CONTROL_DATA 725
user_field 337
user_field_list 366
userid tag in VM/ESA 533

V

validation of data reception 78
values
integers 641
pseudonym 13, 641
variables
characteristics, table 642
in VM/ESA extension routines 576
integer values 642
lengths 649
OS/2 additions 443
pseudonym 13
types 649
types and lengths, table 650
viewing conversation characteristics 34
VM/ESA
conformance classes 528
documentation 526—588
errors 538
extension calls
Extract_Conversation_LUWID (XCECL) 544
Extract_Conversation_Security_User_ID (XCECSU) 546
Extract_Conversation_Workunitid (XCECWU) 548
Extract_Local_Fully_Qualified_LU_Name (XCELFQ) 550
Extract_Remote_Fully_Qualified_LU_Name (XCERFQ) 552
Extract_TP_Name (XCETPN) 554
Identify_Resource_Manager (XCIDRM) 555
Set_Client_Security_User_ID (XCSCUI) 559
Set_Conversation_Security_Password (XCSCSP) 562, 579

VM/ESA *(continued)*
extension calls *(continued)*
Set_Conversation_Security_Type (XCSCST) 564
Set_Conversation_Security_User_ID (XCSCSU) 566, 579
Signal_User_Event (XCSUE) 568
Terminate_Resource_Manager (XCTRRM) 570
Wait_on_Event (XCWOE) 571
HELP Facility, using 588
languages supported 528
reference publications 527
VMCONINPUT system event (VM/ESA) 585
VMCPIC system event (VM/ESA) 585
VMLIB TXTLIB (VM/ESA) 529
multitasking
communications directory 532
Coordinated Resource Recovery 548, 579
work unit 527, 548, 578

W

Wait_For_Completion (CMWCMP) 366
Wait_For_Conversation (CMWAIT) 369
Windows 95
conformance classes
work unit (CMS) 527, 548, 578, 582

X

XCECL (Extract_Conversation_LUWID) call on VM/ESA 544
XCECST (Extract_Conversation_Security_Type)
AIX call 396
OS/2 call 472, 611
XCECWU (Extract_Conversation_Workunit_ID) call on VM/ESA 548
XCELFQ (Extract_Local_Fully_Qualified_LU_Name)
VM/ESA call 550
XCERFQ (Extract_Remote_Fully_Qualified_LU_Name)
VM/ESA call 552
XCETPN (Extract_TP_Name) call on VM/ESA 554
XCINCT (Initialize_Conv_For_TP)
OS/2 call 474, 613
XCMDSI (Delete_CPIC_Side_Information)
OS/2 call 455, 602
XCMESI (Extract_CPIC_Side_Information)
OS/2 call 457, 604
XCMSSI (Set_CPIC_Side_Information)
OS/2 call 460, 607
XCSCSP (Set_Conversation_Security_Password)
AIX call 399
OS/2 call 476, 615
XCSCST (Set_Conversation_Security_Type)
AIX call 401
OS/2 call 477, 616
XCSCSU (Set_Conversation_Security_User_ID)
AIX call 403
OS/2 call 478, 617
XCSUE (Signal_User_Event) call on VM/ESA 568

Communicating Your Comments to IBM

Common Programming Interface
Communications
CPI-C Reference
Version 2.1

Publication No. SC26-4399-09

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

Feel free to comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. However, the comments you send should pertain to only the information in this manual and the way in which the information is presented. To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

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

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

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

  United States and Canada: **1-800-227-5088**
- If you prefer to send comments electronically, use this network ID:
  - IBM Mail Exchange: **USIB2HPD at IBMMAIL**
  - IBMLink: **CIBMORCF at RALVM13**
  - Internet: **USIB2HPD@VNET.IBM.COM**

Make sure to include the following in your note:

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

Help us help you!

Common Programming Interface
Communications
CPI-C Reference
Version 2.1

Publication No. SC26-4399-09

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

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

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

Specific Comments or Problems:

Please tell us how we can improve this book:

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

Name

Address

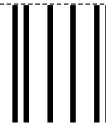
Company or Organization

Phone No.

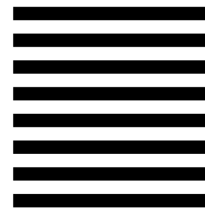
Help us help you!
SC26-4399-09

Cut or Fold
Along Line

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



NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

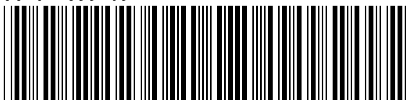
Information Development
Department CGMD
International Business Machines Corporation
PO BOX 12195
RESEARCH TRIANGLE PARK NC 27709-9990

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

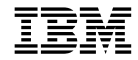
SC26-4399-09

Cut or Fold
Along Line

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.



Spine information:



Common Programming Interface
Communications

CPI-C Reference

Version 2.1