

CICS Transaction Server for z/OS



CICS Application Programming Guide

Version 3 Release 1

CICS Transaction Server for z/OS



CICS Application Programming Guide

Version 3 Release 1

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 701.

This edition applies to Version 3 Release 1 of CICS Transaction Server for z/OS, program number 5655-M15, and to all subsequent versions, releases, and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

© Copyright IBM Corporation 1989, 2010.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Preface	xix
What this book is about	xix
Who should read this book	xix
Summary of changes	xxi
Changes for CICS Transaction Server for z/OS, Version 3 Release 1	xxi
Changes for CICS Transaction Server for z/OS, Version 2 Release 3	xxi
Changes for CICS Transaction Server for z/OS, Version 2 Release 2	xxi
Changes for CICS Transaction Server for z/OS, Version 2 Release 1	xxi
Changes for CICS Transaction Server for OS/390, Version 1 Release 3	xxii

Part 1. Writing CICS Applications 1

Chapter 1. What is a CICS application?	3
CICS programs, transactions and tasks	3
Chapter 2. CICS programming	5
CICS programming commands	5
System programming commands	6
EXEC interface block (EIB)	6
Translation	6
Translator Options	6
Testing for CICS	7
CICS programming roadmap	7
Chapter 3. Language Environment	9
Language Environment services	10
Using Language Environment Abend-handling	12
User-written Language Environment condition handlers	12
Managing Language Environment storage	13
Mixing languages in Language Environment	13
Using Dynamic Link Libraries and Language Environment	15
Defining runtime options for Language Environment	16
Runtime options in child enclaves	18
CEEEXITA coding	18
Determining which options were used	19
Writing a CEEBINT initialization exit for Language Environment	19
Chapter 4. Programming in COBOL	21
COBOL programming considerations	22
COBOL programming restrictions	23
Language Environment CBLPSHPOP option	25
Using the DL/I CALL interface	25
Considerations for VS COBOL II programs	26
Using based addressing with COBOL	27
Using WITH DEBUGGING MODE	28
Calling subprograms from COBOL	28
Rules for calling subprograms	30
Flow of control between programs and subprograms	32
Using the COBOL2 and COBOL3 translator options	34
Literals intervening in blank lines	35
Lower case characters	35
Sequence numbers containing any character	35

REPLACE statement	35
Batch compilation	36
Nested programs	38
Reference modification	41
Global variables	42
Comma and semicolon as delimiters	42
Symbolic character definition	42
Chapter 5. Programming in C and C++	43
C and C++ programming considerations	43
XPLink considerations for C and C++ programming	46
Passing arguments in C or C++.	49
Accessing the EIB.	50
Naming EIB fields.	50
Locale support for C and C++	51
Programming in C++.	51
Restrictions	51
Chapter 6. Programming in PL/I	53
PL/I programming restrictions	53
Language Environment considerations for PL/I applications	54
Chapter 7. Programming in Assembler	57
Language Environment considerations for Assembler applications	58
Calling Assembler programs	60

Part 2. Preparing applications to run 63

Chapter 8. Translation and compilation	67
The integrated CICS translator	67
Using the integrated CICS translator	68
Specifying CICS translator options.	68
The translation process.	69
The CICS-supplied translators	72
Dynamic invocation of the separate translator	72
Using a CICS translator	73
Defining translator options.	74
Translator options.	75
Translator options table.	85
Using COPY statements	86
The CICS-supplied interface modules	86
The EXEC interface modules.	86
The CPI Communications interface module	86
The SAA Resource Recovery interface module	86
Using the EXEC interface modules	87
COBOL	88
PL/I	88
C and C++	89
Assembler language	89
EXAMPLE Assembler language PROGRAM using LEASM.	89
Chapter 9. Installing application programs.	99
Program installation roadmap	99
Preparing for program installation	100
Defining MVS residence and addressing modes	100
Establishing a program's addressing mode	101

CICS address space considerations	101
Making programs permanently resident	102
Running applications in the link pack area	102
Running application programs in the RDSAs	103
Assembler	103
C and C/++.	104
COBOL	104
PL/I	105
Using BMS map sets in application programs	105
Using the CICS-supplied procedures to install application programs	106
Installing programs in load library secondary extents	108
Including the CICS-supplied interface modules	108
Installing assembler language application programs	109
Installing COBOL application programs	110
Sample JCL to install COBOL application programs	111
Installing PL/I application programs	114
Sample JCL to install PL/I application programs	114
PL/I procedure with an integrated translator	115
Installing C application programs	117
Sample JCL to install C application programs	118
Using your own job streams	120
Translator requirements	120
Online programs that use EXEC CICS or EXEC DLI commands	120
Online programs that use the CALL DLI interface	122
Batch or BMP programs that use EXEC DLI commands	123
Batch or BMP programs that use DL/I CALL commands	123
Chapter 10. Installing map sets and partition sets	125
Installing map sets	126
Types of map sets	126
Installing physical map sets	128
Installing symbolic description map sets	129
Installing physical and symbolic description maps together	131
Installing partition sets	133
Defining programs, map sets, and partition sets to CICS	134

Part 3. Application design 135

Chapter 11. Application design	139
Pseudoconversational and conversational design	140
Terminal interruptibility	142
How tasks are started	142
Which transaction?	143
Separating business and presentation logic	146
Multithreading: Reentrant, quasi-reentrant and threadsafe programs	147
Quasi-reentrant application programs	147
Threadsafe programs	149
OPENAPI programs	153
Using the FORCEQR system initialization parameter	155
Non-reentrant programs	155
Storing data within a transaction	156
Transaction work area (TWA)	156
User storage	156
COMMAREA in LINK and XCTL commands	157
Channels in LINK and XCTL commands	158
Program storage	158

Temporary storage	158
Intrapartition transient data	160
GETMAIN SHARED command	160
Your own data sets	160
Lengths of areas passed to CICS commands	162
LENGTH options.	162
Journal records	162
Data set definitions	162
Recommendation	162
Minimizing errors.	163
Protecting CICS from application errors	163
Testing applications.	163
Non-terminal transaction security.	164
 Chapter 12. Design for performance	165
Program size	165
Virtual storage.	166
Reducing paging effects	167
Exclusive control of resources	169
Operational control	170
Operating system waits	171
The NOSUSPEND option	171
Efficient data operations	172
Efficient database operations	172
Efficient data set operations.	172
Efficient browsing (in non-RLS mode)	174
Efficient logging	175
Efficient sequential data set access	175
Efficient terminal operations.	176
Length of the data stream sent to the terminal	176
Basic mapping support considerations	176
Page-building and routing operations	179
Requests for printed output	181
Additional terminal control considerations.	181
 Chapter 13. Sharing data across transactions	183
Using the common work area (CWA)	183
Protecting the CWA.	184
Using the TCTTE user area (TCTUA)	186
Using the COMMAREA in RETURN commands	187
Using a channel on RETURN commands.	187
Using the display screen to share data	188
 Chapter 14. Enhanced inter-program data transfer: channels as modern-day COMMAREAs	189
Channels: quick start	189
Containers and channels.	189
Basic examples	190
Using channels: some typical scenarios	192
One channel, one program	192
One channel, several programs (a component)	193
Several channels, one component	193
Multiple interactive components	194
Creating a channel	195
The current channel	196
Current channel example, with LINK commands	196

Current channel example, with XCTL commands	198
Current channel: START and RETURN commands	199
The scope of a channel	200
Scope example, with LINK commands	200
Scope example, with LINK and XCTL commands	202
Discovering which containers a program's been passed	204
Discovering which containers were returned from a link	204
CICS read only containers	204
Designing a channel: best practices.	205
Constructing and using a channel: an example.	206
Channels and BTS activities	207
Context	208
Using channels from JCICS.	209
Dynamic routing with channels	209
Data conversion	210
Why is data conversion needed?	210
Preparing for code page conversion with channels	210
Data conversion with channels	212
Benefits of channels	215
Migrating from COMMAREAs to channels	216
Migration of existing functions	216
Migration to the new function	216
Chapter 15. Affinity	221
Types of affinity	222
Inter-transaction affinity	222
Transaction-system affinity	222
Programming techniques and affinity	223
Safe techniques	223
Unsafe techniques	223
Suspect techniques.	224
Recommendations	224
Safe programming to avoid affinity	224
The COMMAREA	225
The TCTUA	226
Using ENQ and DEQ commands with ENQMODEL resource definitions	228
BTS containers	229
Unsafe programming for affinity	229
Using the common work area	229
Using GETMAIN SHARED storage	230
Using the LOAD PROGRAM HOLD command	231
Sharing task-lifetime storage	232
Using the WAIT EVENT command	234
Using ENQ and DEQ commands without ENQMODEL resource definitions	235
Suspect programming for affinity	236
Using temporary storage	236
Using transient data	239
Using the RETRIEVE WAIT and START commands	240
Using the START and CANCEL REQID commands	241
Using the DELAY and CANCEL REQID commands	243
Using the POST and CANCEL REQID commands	244
Detecting inter-transaction affinities	246
Inter-transaction affinities caused by application generators	246
Duration and scope of inter-transaction affinities	246
Affinity transaction groups	247
Relations and lifetimes	247

Chapter 16. Recovery design	255
Journaling	255
Journal records	255
Journal output synchronization.	255
Syncpointing	257
Chapter 17. Dealing with exception conditions	261
Default CICS exception handling	261
Handling exception conditions by in-line code	262
How to use the RESP and RESP2 options	262
An example of exception handling in C	263
An example of exception handling in COBOL	264
Modifying default CICS exception handling	265
Using the HANDLE CONDITION command	267
RESP and NOHANDLE options	268
How CICS keeps track of what to do	268
Using the HANDLE CONDITION ERROR command.	269
Using the IGNORE CONDITION command	269
Using the HANDLE ABEND command.	270
Using PUSH HANDLE and POP HANDLE commands	271
Chapter 18. Abnormal termination recovery	273
Creating a program-level abend exit	274
Retrying operations	275
Trace	276
Trace entry points	277
Monitoring	277
Dump	278
Chapter 19. The QUERY SECURITY command	281
Using the QUERY SECURITY command	281
Security protection at the record or field level	281
CICS-defined resource identifiers.	282
SEC system initialization parameter.	282
Programming hints	282

Part 4. Data Management 283

Chapter 20. An overview of file control.	287
VSAM data sets	287
Key-sequenced data set (KSDS)	287
Entry-sequenced data set (ESDS)	288
Relative record data set (RRDS)	288
Empty data sets	289
VSAM alternate indexes	289
Accessing files in RLS mode	289
BDAM data sets	290
CICS shared data tables	291
Coupling facility data tables	292
Coupling facility data table models	293
Techniques for sharing data.	294
Using CICS commands to read records	296
Direct reading (using READ command)	297
Sequential reading (browsing)	298
Skip-sequential processing	300
Using CICS commands to update records	301

Using CICS commands to delete records	302
Deleting single records	302
Deleting groups of records (generic delete)	303
Read integrity	303
Using CICS commands to add records	303
Adding to a KSDS	303
Adding to an ESDS	304
Adding to an RRDS	304
Records that are already locked	304
Specifying record length	304
Sequential adding of records (WRITE MASSINSERT command)	304
File control command options	305
The RIDFLD option	305
The INTO and SET options	305
The FROM option	306
The TOKEN option	306
KEYLENGTH option for remote data sets.	306
Transaction deadlocks.	307
VSAM-detected deadlocks (RLS only)	308
Rules for avoiding deadlocks	309
Chapter 21. File control—VSAM considerations	311
VSAM record identification	311
Key.	311
Relative byte address (RBA) and relative record number (RRN)	311
Locking of VSAM records in recoverable files	312
Update locks and delete locks (non-RLS mode only)	312
RLS Record level locking	313
Exclusive locks and shared locks.	313
Conditional VSAM file update requests	316
File control implementation of NOSUSPEND	316
CICS locking for writing to ESDS.	316
Chapter 22. File control—BDAM considerations	319
Identifying BDAM records	319
Block reference subfield	319
Physical key subfield	319
Deblocking argument subfield	319
Browsing records from BDAM data sets	320
Adding records to BDAM data sets	321
Updating records from BDAM data sets	322
Chapter 23. Database control	323
DL/I databases	323
DB2 databases	323
Requests to DB2.	323
Chapter 24. CICS documents	325
Using the DOCUMENT programming interface.	325
Creating a document	325
Setting up document templates	326
Templates in a partitioned data set	326
Templates in a CICS program	327
Templates in CICS files, z/OS UNIX System Services HFS files, temporary storage, or transient data	327
Creating templates in exit programs.	328

Programming with documents and document templates	329
Symbols and symbol lists	329
Setting symbol values	330
Embedded template commands	332
Using templates in your application	333
The lifespan of a document	334
Constructing a document	336
Using Bookmarks	338
Code page conversion for documents	339
Chapter 25. Named counter servers	341
The named counter fields	341
Named counter pools	342
Named counter options table	342
Using the named counter EXEC interface	343
Using the named counter CALL interface	345
Application programming considerations	345
Syntax	347
Return codes	355
Named counter recovery	358

Part 5. Data Communication. 361

Chapter 26. Terminal control.	365
Terminal access method support	366
Terminal control commands	366
Send/receive mode	367
Speaking out of turn	368
Interrupting	369
Terminal waits	369
Using data transmission commands	370
What you get on a RECEIVE	370
Device control commands	371
Terminal device support	372
Finding out about your terminal	375
EIB feedback on terminal control operations	377
Using VTAM	378
Chaining input data	378
Chaining output data	378
Handling logical records	379
Response protocol	379
Using function management headers	380
Preventing interruptions (bracket protocol)	380
Using sequential terminal support	381
Coding considerations for sequential terminals	382
Using TCAM	383
Coding for the TCAM/DCB interface	383
Using batch data interchange	383
Destination selection and identification	385
Definite response	385
Waiting for function completion	385
Chapter 27. The 3270 family of terminals	387
History of the 3270	387
Screen fields	388
Personal computers	388

The 3270 buffer	390
The output datastream	390
3270 write commands	390
3270 display fields	392
Display characteristics	392
3270 field attributes	392
Protection	393
Modification	393
Intensity	394
Base color	394
Extended attributes	394
Orders in the data stream	395
The start field order	395
The modify field order	396
The set buffer address order	397
The set attribute order	398
Outbound data stream sample	398
Input from a 3270 terminal	400
Data keys	400
Keyboard control keys	401
Attention keys	401
Reading from a 3270 terminal	403
Inbound field format	403
Input data stream example	404
Unformatted mode	404
 Chapter 28. CICS support for printing	 407
Formatting for CICS printers	407
CICS 3270 printers	408
CICS 3270 printer options	409
PRINT option and print control bit	409
ERASE option	410
Line width options: L40, L64, L80, and HONEOM	410
NLEOM option	410
FORMFEED	411
PRINTERCOMP option	412
Non-3270 CICS printers	412
SCS input	413
Determining the characteristics of a CICS printer	413
BMS page size, 3270 printers	414
Supporting multiple printer types	414
Using CICS printers	415
Printing with a START command	415
Printing with transient data	415
Printing with BMS routing	417
Using Non-CICS printers	417
Formatting for non-CICS printers	417
Non-CICS printers: Delivering the data	417
Programming for non-CICS printers	418
Notifying the print application	419
Printing display screens	420
CICS print key	420
ISSUE PRINT and ISSUE COPY	420
Hardware print key	421
BMS screen copy	421

Chapter 29. CICS interface to JES	423
Using the CICS interface to JES	425
Spool interface restrictions	425
Creating output spool files	425
Using the MVS internal reader.	426
Reading input spool files	426
JES exits	427
Identifying spool files	427
Examples of SPOOL commands	430
COBOL	430
PL/I	431
C	431
ASSEMBLER	432
 Chapter 30. CICS intercommunication	 433
Design considerations	433
Programming language	434
Transaction routing	434
Function shipping	434
Distributed program link (DPL)	435
Using the distributed program link function	436
Examples of distributed program link	437
Programming considerations for distributed program link	442
Asynchronous processing	446
Distributed transaction processing (DTP)	446
Common Programming Interface Communications (CPI Communications)	446
External CICS interface (EXCI)	447

Part 6. Basic Mapping Support (BMS) 449

Chapter 31. Basic mapping support	453
BMS support levels	453
Minimum BMS	453
Standard BMS	453
Full BMS	454
A BMS output example	455
 Chapter 32. Creating the map	 459
Defining map fields: DFHMDF	459
Defining the map: DFHMDI	461
Defining the map set: DFHMSD	462
Writing BMS macros	463
Assembling the map	465
Physical and symbolic map sets	465
The SDF II alternative	466
Grouping maps into map sets	466
The Application Data Structure (ADS)	467
Using complex fields	467
Composite fields: the GRPNAME option	468
Repeated fields: the OCCURS option	469
Block data	469
Support for non-3270 terminals	470
Output considerations for non-3270 devices	470
Differences on input	471
Special options for non-3270 terminals	472
Device-dependent maps	472

Device dependent support: DDS	473
Finding out about your terminal	475
Chapter 33. Sending BMS mapped output	477
Acquiring and defining storage for the maps.	477
BASE and STORAGE options	478
Initializing the output map	478
Moving the variable data to the map	479
Setting the display characteristics	479
Changing the attributes	480
Attribute value definitions: DFHBMSCA	481
Chapter 34. Using the SEND MAP command	483
SEND MAP control options	483
Other BMS SEND options: WAIT and LAST.	484
Merging the symbolic and physical maps	484
MAPONLY option	484
DATAONLY option	485
The SEND CONTROL command.	485
Building the output screen	485
What you start with	486
What is sent	486
Where the values come from	486
Outside the map	487
Using GDDM and BMS	488
Positioning the cursor	488
Sending invalid data and other errors	489
Output disposition options: TERMINAL, SET, and PAGING	489
Using SET	490
Chapter 35. Receiving mapped data	493
An input-output example	493
The symbolic input map	495
Programming mapped input.	496
Using the RECEIVE MAP command	496
Getting storage for mapped input.	497
Formatted screen input	497
Modified data	498
Upper case translation	499
Using the attention identifier	499
Using the HANDLE AID command	499
Finding the cursor	500
Processing the mapped input	500
Handling input errors	501
Flagging errors	501
Saving the good input	502
Rechecking.	502
Sending mapped output after mapped input.	503
MAPFAIL and other exceptional conditions	504
EOC condition.	504
Formatting other input.	505
Chapter 36. BMS logical messages	507
Building logical messages	507
The SEND PAGE command	508
RETAIN and RELEASE	508

The AUTOPAGE option	510
Terminal operator paging: the CSPG transaction	510
Logical message recovery	511
Chapter 37. Cumulative output — the ACCUM option	513
Floating maps: how BMS places maps using ACCUM	513
Page breaks: BMS overflow processing	514
Map placement rules	515
ASSIGN options for cumulative processing	517
Input from a composite screen.	517
Performance considerations.	517
Minimizing path length.	517
Reducing message lengths	518
Chapter 38. Text output.	521
The SEND TEXT command.	521
Text logical messages	521
Text pages	522
Text lines	523
Header and trailer format.	524
SEND TEXT MAPPED and SEND TEXT NOEDIT	525
Chapter 39. Message routing	527
Message destinations	527
Eligible terminals.	528
Destinations specified with OPCLASS only	528
OPCLASS and LIST omitted	529
Route list provided	529
Route list format	530
Message delivery	531
Undeliverable messages	532
Recoverable messages	532
Message identification	533
Programming considerations with routing	534
Routing and page overflow	534
Routing with SET	535
Interleaving a conversation with message routing	535
Chapter 40. The MAPPINGDEV facility	537
SEND MAP with the MAPPINGDEV option	537
RECEIVE MAP with the MAPPINGDEV option.	538
Sample assembler MAPPINGDEV application	539
Chapter 41. Partition support	541
Uses for partitioned screens	542
Scrolling	542
Data entry	542
Lookaside	542
Data comparison.	543
Error messages	543
Partition definition	543
3290 character size.	544
Establishing partitioning	544
Partition options for BMS SEND commands.	545
Determining the active partition	545
Partition options for BMS RECEIVE commands	546

ASSIGN options for partitions	546
Partitions and logical messages	546
Partitions and routing	547
Attention identifiers and exception conditions	547
Terminal sharing	548
Chapter 42. Support for special hardware	549
Logical device components	549
Defining logical device components	549
Sending data to a logical device component.	550
LDCs and logical messages	550
LDCs and routing	550
10/63 magnetic slot reader	551
Field selection features	551
Trigger field support	551
Cursor and pen-detectable fields	552
Selection fields	553
Attention fields	553
BMS input from detectable fields	553
Outboard formatting	554

Part 7. CICS management functions 555

Chapter 43. Interval control	557
Expiration times	558
Request identifiers	559
Chapter 44. Task control	561
Controlling sequence of access to resources	562
Chapter 45. Program control.	565
Program linking	566
Application program logical levels	566
Link to another program expecting return.	566
Passing data to other programs	567
COMMAREA	567
Channels	569
INPUTMSG.	569
Using mixed addressing modes	571
Using LINK to pass data	572
Using RETURN to pass data	574
Chapter 46. Storage control	577
Overview of CICS storage protection and transaction isolation	578
Storage protection	578
Transaction isolation	579
Defining the storage key for applications	580
System-wide storage areas	581
Task lifetime storage	581
Program working storage specifically for exit and PLT programs	581
Passing data by a COMMAREA	581
The GETMAIN command	582
Selecting the execution and storage key	583
User-key applications	584
CICS-key applications	584
Using transaction isolation	587

MVS subspaces	589
Subspaces and basespaces for transactions	589
The common subspace and shared storage	590
Chapter 47. Transient data control	593
Intrapartition transient data queues	593
Extrapartition queues	594
Indirect queues	595
Automatic transaction initiation (ATI)	595
Chapter 48. Temporary storage control	597
Temporary storage queues	597
Typical uses of temporary storage control	598

Part 8. Testing and debugging applications 601

Chapter 49. Testing applications	603
Preparing the application for testing	604
Preparing the system for testing	604

Chapter 50. Execution diagnostic facility (EDF)	607
Restrictions when using EDF	608
OPEN TCBs and EDF	610
Parameter list stacking	610
Security considerations	610
What does EDF display?	611
The header	611
The body	612
Using EDF	618
Interrupting program execution	619
Using EDF in single-screen mode	620
Using EDF in dual-screen mode	622
EDF and remote transactions	622
EDF and non-terminal transactions	622
EDF and DTP programs	623
Stopping EDF	624
Overtyping to make changes	624
EDF responses	626
Using EDF menu functions	626

Chapter 51. Temporary storage browse (CEBR)	633
Using the CEBR transaction	633
What does the CEBR transaction display?	635
The header	635
The command area	635
The body	635
The message line	635
Using the CEBR function keys	636
Using the CEBR commands	637
Using the CEBR transaction with transient data	639

Chapter 52. Command-level interpreter (CECI)	641
What does CECI display?	641
The command line	641
The status line	642
The body	646

#

#

The message line	646
CECI options on function keys.	646
Using CECI.	648
Making changes	649
Using the CECI function keys	650
Expanded area	650
Variables.	650
The EXEC interface block (EIB)	652
Error messages display	652
Saving commands	653
How CECI runs	654
CECI sessions	654
Abends	654
Exception conditions	655
Program control commands.	655
Terminal sharing	655
Shared storage: ENQ commands without LENGTH option	655
 Chapter 53. Using debuggers with CICS applications	657
Debugging profiles	658
Using debugging profiles to select programs for debugging	659
Using generic parameters in debugging profiles	661
 Chapter 54. Debugging CICS applications from a workstation.	663
Preparing to debug applications from a workstation	663
 Chapter 55. Using Debug Tool with CICS applications.	665
About Debug Tool	665
Preparing to debug applications with Debug Tool	665

Part 9. Appendixes 667

Appendix A. Using the phonetic conversion subroutine DFHPHN	669
 Appendix B. Migration for OS/VS COBOL programs	671
Conversion to Enterprise COBOL	671
Based addressing	671
Artificial assignments	673
 Bibliography	675
The CICS Transaction Server for z/OS library	675
The entitlement set	675
PDF-only books	675
Other CICS books	677
Books from related libraries.	677
DL/I	677
DB2	677
Screen definition facility II (SDF II)	677
Common programming interface	678
Common user access	678
Programming languages	678
Teleprocessing Network Simulator (TPNS)	678
Language Environment:	678
Miscellaneous books	679
Determining if a publication is current	679

Accessibility	681
Index	683
Notices	701
Trademarks.	702
Sending your comments to IBM	703

Preface

What this book is about

This book gives **guidance** about the development of procedural application programs that use the CICS® EXEC application programming interface to access CICS services and resources; it complements the **reference** information in the *CICS Application Programming Reference* manual. For guidance information on debugging such CICS applications, see the *CICS Problem Determination Guide*. For guidance on developing application programs using the Java™ language, see *Java Applications in CICS*, and for guidance on using the CICS OO classes, see *CICS C++ OO Class Libraries*.

Who should read this book

This book is mainly for experienced application programmers. Those who are relatively new to CICS should be able to understand it. If you are a system programmer or system analyst, you should still find it useful.

What you need to know to understand this book

You must be able to program in COBOL, C, C++, PL/I, or assembler language, and have a basic knowledge of CICS application programming, at the *Designing and Programming CICS Applications* level.

How to use this book

Read the parts covering what you need to know. (Each part has a full table of contents to help you find what you want.) The book is a guide, not a reference manual. On your first reading, it probably helps to work through any one part of it more or less from start to finish.

Notes on terminology

API refers to the CICS command-level application programming interface unless otherwise stated.

ASM is sometimes used as the abbreviation for assembler language.

MVS™ refers to the operating system, which can be either an element of z/OS®, OS/390®, or MVS/Enterprise System Architecture System Product (MVS/ESA SP).

VTAM® refers to ACF/VTAM.

In the sample programs described in this book, the dollar symbol (\$) is used as a national currency symbol and is assumed to be assigned the EBCDIC code point X'5B'. In some countries a different currency symbol, for example the pound symbol (£), or the yen symbol (¥), is assigned the same EBCDIC code point. In these countries, the appropriate currency symbol should be used instead of the dollar symbol.

What is not covered in this book

Guidance for usage of the CICS Front End Programming Interface is not discussed in this book. See the *CICS Front End Programming Interface User's Guide* for background information about FEPI design considerations and programming information about its API.

Guidance for usage of the EXEC CICS WEB commands is not discussed in this book. See the *CICS Internet Guide* for this information.

Guidance for the use of object oriented programming languages and techniques is not included in this book. For guidance on developing application programs using the Java language, see *Java Applications in CICS*, and for guidance on using the CICS OO classes, see *CICS C++ OO Class Libraries*.

Summary of changes

This book is based on the CICS Application Programming Guide for CICS Transaction Server for z/OS, Version 2 Release 3. Changes from that edition are marked by vertical bars in the left margin.

Changes for CICS Transaction Server for z/OS, Version 3 Release 1

The more significant changes for this edition are:

- Technical changes:
 - Enhanced inter-program data transfer: channels as modern-day COMMAREAs is a new chapter that describes how programs can use “channels” and “containers” to exchange data.

Structural changes :

- The chapter “Writing Web-aware application programs” has been moved to the *CICS Internet Guide*.

Changes for CICS Transaction Server for z/OS, Version 2 Release 3

The more significant changes for this edition are:

- Technical changes:
 - Chapter 53, “Using debuggers with CICS applications,” on page 657, Chapter 54, “Debugging CICS applications from a workstation,” on page 663 and Chapter 55, “Using Debug Tool with CICS applications,” on page 665 have been added.

There are no significant structural changes.

Changes for CICS Transaction Server for z/OS, Version 2 Release 2

The more significant changes for this edition are:

- Technical changes:
 - The use of Language Environment® is assumed in all programming guidance information.
- Structural changes :
 - Information in “Creating a document” on page 325, describing the use of document templates, has been expanded to incorporate information that was previously in the *CICS Internet Guide*

Changes for CICS Transaction Server for z/OS, Version 2 Release 1

The following new function has been added:

- Support for an integrated translator. Some compilers can now interpret CICS commands and translate them without the need for a separate translation step, see “The integrated CICS translator” on page 67.

Changes have been made to titles and headings throughout the book, to make them more meaningful, particularly when the book is explored online in the CICS Information Center. Links and cross references have been improved.

- Part 1, Writing CICS applications, has been expanded to include an introduction to basic CICS concepts and an application development roadmap. See “CICS programming roadmap” on page 7.

- Part 2 in the previous edition, Object Oriented programming in CICS, has been removed. For guidance on developing application programs using the Java language, see the *Java Applications in CICS* component of the CICS Information Center, and for guidance on using the CICS OO classes, see *CICS C++ OO Class Libraries*.
- A new part 2 has been introduced, to bring together all information needed to translate and compile CICS applications. Chapters describing the installation of CICS programs and maps have been moved here from the *CICS System Definition Guide*.
- Part 3, Application design, now separates general application design concepts, see Chapter 11, “Application design,” on page 139, from application design for performance, see Chapter 12, “Design for performance,” on page 165.
- BMS has been restructured into separate chapters, to form a new Part 6.
- The previous Appendix 1, mapping EXEC CICS commands to obsolete CICS macros has been removed.

Changes for CICS Transaction Server for OS/390, Version 1 Release 3

The following significant changes were made for this edition.

- The addition of the JCICS Java classes to access CICS services from Java application programs. Now moved to the *Java Applications in CICS* component of the CICS Information Center
- Support for running CICS Java programs using the VisualAge® for Java, Enterprise Edition for OS/390. Now moved to the *Java Applications in CICS* component of the CICS Information Center.
- Support for running CICS Java programs using the CICS Java Virtual Machine (JVM). Now moved to the *Java Applications in CICS* component of the CICS Information Center.
- The addition of sysplex-wide ENQ and DEQ. See “Using ENQ and DEQ commands with ENQMODEL resource definitions” on page 228.
- The addition of support for coupling facility data tables (CFDT). See “Coupling facility data tables” on page 292.
- Support for named counter servers. See Chapter 25, “Named counter servers,” on page 341.
- Support for documents, and the EXEC CICS DOCUMENT commands. See Chapter 24, “CICS documents,” on page 325.
- The programming considerations section has been reorganized into separate chapters for each supported language, including new chapters for OO and Java support in CICS.

Part 1. Writing CICS Applications

Chapter 1. What is a CICS application?	3
CICS programs, transactions and tasks	3
Chapter 2. CICS programming	5
CICS programming commands	5
System programming commands.	6
EXEC interface block (EIB)	6
Translation	6
Translator Options	6
Testing for CICS	7
CICS programming roadmap	7
Chapter 3. Language Environment	9
Language Environment services	10
Using Language Environment Abend-handling	12
User-written Language Environment condition handlers	12
Managing Language Environment storage	13
Mixing languages in Language Environment	13
Using Dynamic Link Libraries and Language Environment	15
Defining runtime options for Language Environment	16
Runtime options in child enclaves	18
CEEEXITA coding	18
Determining which options were used	19
Writing a CEEBINT initialization exit for Language Environment	19
Chapter 4. Programming in COBOL	21
COBOL programming considerations	22
COBOL programming restrictions	23
Language Environment CBLPSHPOP option	25
Using the DL/I CALL interface	25
Considerations for VS COBOL II programs	26
Using based addressing with COBOL	27
Using WITH DEBUGGING MODE	28
Calling subprograms from COBOL	28
Rules for calling subprograms	30
Translation	30
Compilation	30
Link-editing	30
CICS CSD entries without program autoinstall	30
Return from subprogram	31
Language of subprogram	31
Contents of subprogram	31
Passing parameters to subprogram	31
Storage	31
CICS condition, AID and abend handling	32
Location of subprogram	32
Flow of control between programs and subprograms	32
Using the COBOL2 and COBOL3 translator options	34
Literals intervening in blank lines	35
Lower case characters	35
Sequence numbers containing any character	35
REPLACE statement	35
Batch compilation	36

Nested programs	38
Integrated CICS translator.	38
Translator action	38
Comments in translator input.	39
Nesting: what the application programmer must do	39
An example of a nested program	39
Reference modification	41
Global variables	42
Comma and semicolon as delimiters	42
Symbolic character definition.	42
Chapter 5. Programming in C and C++	43
C and C++ programming considerations	43
XPLink considerations for C and C++ programming	46
XPLink, and the X8 and X9 TCBs	47
Writing C and C++ programs, which are to be compiled with the XPLINK option, for the CICS environment	48
Passing control between XPLink and non-XPLink objects	48
Changing CICS definitions to obtain CICS support for objects compiled with the XPLINK option	48
Global User exits and XPLink	48
Passing arguments in C or C++.	49
Accessing the EIB.	50
Naming EIB fields.	50
Data types in EIB fields.	50
Locale support for C and C++	51
Programming in C++.	51
Restrictions	51
Chapter 6. Programming in PL/I	53
PL/I programming restrictions	53
Language Environment considerations for PL/I applications	54
Chapter 7. Programming in Assembler	57
Language Environment considerations for Assembler applications	58
Calling Assembler programs	60

Chapter 1. What is a CICS application?

An application is a collection of related programs that together perform a business operation, such as processing a product order or preparing a company payroll. CICS applications execute under CICS control, using CICS services and interfaces to access programs and files.

CICS is a transaction processing subsystem. That means that it provides services for you to run applications online, by request, at the same time as many other users are submitting requests to run the same applications, using the same files and programs. CICS manages the sharing of resources; integrity of data and prioritization of execution, with fast response.

CICS applications are traditionally run by submitting a **transaction** request. Execution of the transaction consists of running one or more **application programs** that implement the required function. In CICS documentation you may find CICS application programs sometimes simply called **programs**, and sometimes the term transaction is used to imply the processing done by the application programs.

CICS applications can also take the form of Enterprise JavaBeans. You can find out more about this form of programming in *Java Applications in CICS* in the CICS Information Center.

You should note that the term **transaction** is now used extensively in the IT industry to describe a **unit of recovery** or what CICS calls a **unit of work**. This is typically a complete operation that is recoverable; it can be committed or backed out as an entirety as a result of programmed command or system failure. In many cases the scope of a CICS transaction is also a single unit of work, but you should be aware of the difference in meaning when reading CICS documentation.

CICS programs, transactions and tasks

To develop and run CICS applications, you need to understand the relationship between CICS programs, transactions and tasks. These terms are used throughout CICS documentation and appear in many commands.:

Transaction

A transaction is a piece of processing initiated by a single request. This is usually from an end-user at a terminal, but may also be made from a Web page, from a remote workstation program, from an application in another CICS system or triggered automatically at a predefined time. The *CICS Internet Guide* and the *CICS External Interfaces Guide* describe different ways of running CICS transactions.

A single transaction consists of one or more **application programs** that, when run, carry out the processing needed.

However, the term **transaction** is used in CICS to mean both a single event and all other transactions of the same type. You describe each transaction type to CICS with a TRANSACTION resource definition. This definition gives the transaction type a name (the transaction identifier, or TRANSID) and tells CICS several things about the work to be done; such as what program to invoke first, and what kind of authentication is required throughout the execution of the transaction.

You run a transaction by submitting its TRANSID to CICS. CICS uses the information recorded in the TRANSACTION definition to establish the correct execution environment, and starts the first program.

The term **transaction** is now used extensively in the IT industry to describe a **unit of recovery** or what CICS calls a **unit of work**. This is typically a complete operation that is recoverable; it can be committed or backed out as an entirety as a result of programmed command or system failure. In many cases the scope of a CICS transaction is also a single unit of work, but you should be aware of the difference in meaning when reading non-CICS documentation.

Task You will also see the word **task** used extensively in CICS documentation. This word also has a specific meaning in CICS. When CICS receives a request to run a transaction, it starts a new task that is associated with this *one instance* of the execution of the transaction. type. That is, one execution of a transaction, with a particular set of data, usually on behalf of a particular user at a particular terminal. You can also consider it as analogous to a *thread*. When the transaction completes, the task is terminated.

Chapter 2. CICS programming

You write a CICS program in much the same way as you write any other program. You can use COBOL, C, C++, Java, PL/I, or assembler language to write CICS application programs. Most of the processing logic is expressed in standard language statements, but you use CICS commands, or the Java and C++ class libraries to request CICS services.

This book describes the use of the CICS command level programming interface, 'EXEC CICS', that can be used in COBOL, C, C++, PL/I or assembler programs. These commands are defined in detail in the *CICS Application Programming Reference*.

Programming in Java with the JCICS class library is described in the *Java Applications in CICS* component of the CICS Information Center.

Programming in C++ with the CICS C++ classes is described in the *CICS C++ OO Class Libraries* documentation.

For information about writing Web applications to process HTTP/1.0 requests and responses, see the *CICS Internet Guide*.

For further guidance on language use with CICS, see Chapter 4, "Programming in COBOL," on page 21, Chapter 5, "Programming in C and C++," on page 43, Chapter 6, "Programming in PL/I," on page 53..

CICS allows you to use SQL statements, DLI requests, CPI statements, and the CICS Front End Programming Interface (FEPI) commands in your program as well as CICS commands. You need to consult additional manuals for information:

- *SQL Reference* manual and the *Application Programming and SQL Guide* (for SQL)
- *Application Programming:EXEC DLI Commands* manual and the *Application Programming: DL/I Calls* manual (for DL/I)
- *IBM SAA: CPI Reference* manual and the *SAA Common Programming Interface for Resource Recovery Reference* manual (for CPI)
- *CICS Front End Programming Interface User's Guide* (for programming information about FEPI commands)

CICS programming commands

The general format of a CICS command is EXECUTE CICS (or EXEC CICS) followed by the name of the required command and possibly one or more options.

You can write many application programs using the CICS command-level interface without any knowledge of, or reference to, the fields in the CICS control blocks and storage areas. However, you might need to get information that is valid outside the local environment of your application program.

You can use the ADDRESS and ASSIGN commands to access such information. For programming information about these commands, see the *CICS Application Programming Reference* manual.

When using the ADDRESS and ASSIGN commands, various fields can be read but should not be set or used in any other way. This means that you should not use

any of the CICS fields as arguments in CICS commands, because these fields may be altered by the EXEC interface modules.

System programming commands

The INQUIRE, SET, and PERFORM commands allow application programs to access information about CICS resources. The application program can retrieve and modify information for CICS data sets, terminals, system entries, mode names, system attributes, programs, and transactions. These commands plus the spool commands of the CICS interface to JES, are primarily for the use of the system programmer. For programming information, see the *CICS System Programming Reference* manual.

EXEC interface block (EIB)

In addition to the usual CICS control blocks, each task in a command-level environment has a control block known as the EXEC interface block (EIB) associated with it. An application program can access all of the fields in the EIB by name. The EIB contains information that is useful during the execution of an application program, such as the transaction identifier, the time and date (initially when the task is started, and subsequently, if updated by the application program using ASKTIME), and the cursor position on a display device. The EIB also contains information that is helpful when a dump is used to debug a program. For programming information about EIB fields, see the *CICS Application Programming Reference* manual.

Translation

Most compilers (and assemblers) cannot process CICS commands directly. This means that an additional step is needed to convert your program into executable code. This step is called **translation**, and consists of converting CICS commands into the language in which the rest of the program is coded, so that the compiler (or assembler) can understand them.

Some compilers now contain **integrated translators** that can interpret CICS commands and convert them automatically to calls to CICS service routines. If you use one of these compilers, you do not need to perform the translation tasks described in “The translation process” on page 69.

CICS provides a translator program for each of the languages in which you may write, to handle both EXEC CICS and EXEC DLI statements.

Translator Options

You can specify a number of options for the translation process, and you may need to do this for certain types of programs. If you are using EXEC DLI, for example, you need to tell the translator this fact. “Using a CICS translator” on page 73 explains how to specify options, and “Defining translator options” on page 74 defines the options available.

Testing for CICS

Your program can determine whether it is running in CICS in two different ways:

iscics

If you are adapting an existing C language program or writing a new program that is designed to run outside CICS as well as under CICS, the C language `iscics()` function may prove useful. It returns a non-zero value if your program is currently running under CICS, or zero otherwise. This function is an extension to the C library.

DFH3QSS

Your program can call the DFH3QSS program to query the CICS environment and API capability. Link DFH3QSS statically linked into your own application. On return, register 15 addresses a result structure that consists of a half-word length (that includes itself) followed by a reserved half-word (currently zero) followed by a bit string:

- | | |
|--------------|--|
| Bit 0 | When set to 1, this means that the caller is running in a CICS environment (on a CICS-managed TCB or one of its descendants). |
| Bit 1 | When set to 1, this means that the CICS API is available to the caller (in the current PSW key, ASC-mode, AMODE and cross-memory environment). |

The output structure remains accessible as long as the TCB under which the request was issued has not terminated and DFH3QSS itself is still present in virtual storage. Any change of execution state (such as PSW key, ASC-mode, AMODE or cross-memory environment) might affect the availability of the CICS API. Registers are preserved.

CICS programming roadmap

Follow these steps to develop a CICS application that uses the EXEC CICS command level programming interface:

1. Design your application, identifying the CICS resources and services you will use. See Chapter 11, “Application design,” on page 139 and Chapter 12, “Design for performance,” on page 165 for guidance on designing CICS applications.
2. Write your program in the language of your choice, including EXEC CICS commands to request CICS services. See the *CICS Application Programming Reference* for a list of CICS commands.
3. If you are using a compiler that incorporates “The integrated CICS translator” on page 67, you will only need to compile your program, and then install it in CICS, using the process described in “Program installation roadmap” on page 99. Otherwise, you will need to define translator options for your program, using the process described in “Using a CICS translator” on page 73, and then translate and compile your program, and install it in CICS, using the process described in “Program installation roadmap” on page 99.
4. Define your program and related transaction to CICS with PROGRAM resource definitions and TRANSACTION resource definitions as described in the *CICS Resource Definition Guide* .
5. Define any CICS resources that your program uses, such as files, queues or terminals.

6. Make the resources known to CICS using the CEDA INSTALL command described in the *CICS Resource Definition Guide*.
7. Run your program, by entering the transaction identifier at a CICS terminal, or by using any of the methods described in the *CICS External Interfaces Guide* and the *CICS Internet Guide*.

Chapter 3. Language Environment

Language Environment, supplied as an element of z/OS, is designed to provide a common set of runtime libraries to replace the native runtime libraries that were provided with older compilers such as VS COBOL II, OS PL/I and C/370™.

Before the introduction of Language Environment, each of the high-level languages (HLLs) had to provide a separate runtime environment. With Language Environment, you use one runtime environment for your applications, regardless of the programming language or system resource needs, because most system dependencies have been removed. This common environment offers two significant advantages:

1. You can mix all the languages supported by CICS in a single program.
2. The same Language Environment callable services are available to all programs. This means, for example, that:
 - A linked-list created with storage obtained using Language Environment callable services in a PL/I program can be processed later and the storage freed using the callable services from a COBOL routine.
 - The currency symbol to be used on a series of reports can be set in an assembler routine, even though the reports themselves are produced by COBOL programs.
 - System messages from programs written in different languages are all sent to the same output destination.

See the *z/OS: Language Environment Concepts Guide* for more information.

Because of these advantages, high-level language support under CICS depends upon Language Environment.

The CICS programming guidance documentation requires that your CICS system is using the services of Language Environment, which provides a common runtime environment for IBM® implementations of assembler and those high-level languages (HLLs) supported by CICS, namely COBOL, PL/I, C, and C++.

CICS supports application programs compiled using most of the compilers that are supported by Language Environment. For a list of compilers that are supported in this release of CICS Transaction Server for z/OS, see the *CICS Release Guide*.

Most of the compilers supported by CICS and Language Environment are Language Environment-conforming compilers, meaning that programs compiled by these compilers can take advantage of all Language Environment facilities in a CICS region. CICS and Language Environment also support programs compiled by some pre-Language Environment compilers (which are not Language Environment-conforming). However, CICS does not support all the pre-Language Environment compilers which are supported by Language Environment. The following pre-Language Environment compilers are supported by Language Environment, but are unsupported in this release of CICS:

- OS PL/I Versions 1 and 2
- C/370 V1 and V2

The following pre-Language Environment compilers are supported by Language Environment and supported by CICS:

- AD/Cycle C/370 V1R1

- VS COBOL II

Note: Support for OS/VS COBOL programs is now withdrawn. These programs, which had runtime support in CICS Transaction Server for z/OS Version 2, cannot run under CICS Transaction Server for z/OS Version 3. OS/VS COBOL programs must be upgraded to a supported level of COBOL, and recompiled against a level of COBOL compiler supported by CICS.

See Appendix B, “Migration for OS/VS COBOL programs,” on page 671 for notes on converting OS/VS COBOL programs to Enterprise COBOL. The *Enterprise COBOL for z/OS: Compiler and Run-Time Migration Guide* has more detailed information about language differences, and describes facilities to help with conversion.

Applications compiled and linked with pre-Language Environment compilers usually execute successfully using the runtime support provided by Language Environment. They do not usually have to be recompiled or re-link-edited. In some circumstances, you might need to adjust Language Environment runtime options to enable these applications to execute correctly. Refer to the *Language Environment Run-Time Application Migration Guide*, and the *Compiler and Run-Time Migration Guide* for the language in use, for further information. Because these compilers are not Language Environment conforming, programs compiled by these compilers cannot take advantage of all Language Environment facilities in a CICS region.

The native runtime libraries provided with pre-Language Environment compilers are not supported. Language libraries, other than the Language Environment libraries, should not be present in your CICS startup JCL. If, perhaps for commonality with other CICS systems, the JCL for your CICS startup job includes other language libraries, the Language Environment libraries must be above all the other language libraries in the JCL concatenations of the CICS startup job for both STEPLIB and DFHRPL. This ensures that the programs are processed by Language Environment.

When modifying existing application programs, or writing new programs, you must use a compiler supported by Language Environment. This requires that your application programs must be link-edited using the Language Environment SCEELKED library, and this in turn means that the resulting application load module can execute only under Language Environment.

CICS Transaction Server for z/OS Version 3 allows you to produce Language Environment-conforming assembler MAIN programs. For more information about assembler programs, see Chapter 7, “Programming in Assembler,” on page 57.

Language Environment services

The following points apply to all programs using Language Environment under CICS.

Callable services

Language Environment provides callable services, which are classified in the following categories:

Storage services

These allow you to allocate and free storage from the Language Environment heaps.

Error handling services

These provide a common method of obtaining information to enable you to process errors.

Message services

These provide a common method of handling and issuing messages.

Date and time

These allow you to read, calculate, and write values representing the date and time. Language Environment offers unique pattern-matching capabilities that let you process almost any date and time format contained in an input record or produced by operating system services.

National language support

These allow you to customize Language Environment output (such as messages, RPTOPTS reports, RPTSTG reports, and dumps) for a given country.

Locale

These allow you to customize culturally-sensitive output for a given national language, country, and codeset by specifying a locale name.

General

These are a set of callable services that are not directly related to a specific Language Environment function, for example, dump.

Mathematical

These allow you to perform standard mathematical computations.

These services are normally **only** available to programs compiled with Language Environment-conforming compilers. As an exception, VS COBOL II programs can make dynamic calls to the date and time callable services, but they cannot make any other dynamic calls or any static calls to Language Environment callable services.

For further information about the details of these services, see the *z/OS: Language Environment Programming Guide*. For information about the syntax required to call any of the services, see the *z/OS: Language Environment Programming Reference*.

CICS condition and AID handling

Language Environment condition handling does not alter the behavior of applications that use CICS HANDLE CONDITION or HANDLE AID commands. Language Environment is not involved in the handling of CICS-defined exception conditions, which are raised and handled only by CICS. Similarly, AID detection is a CICS function unaffected by Language Environment .

Abend handling

Language Environment Abend handling depends on the use of CICS HANDLE ABEND. See “Using Language Environment Abend-handling” on page 12 for details.

Storage

Language Environment uses storage obtained from CICS for each run-unit. See “Managing Language Environment storage” on page 13 for information about CICS parameters to control this.

Message and dump destinations

When the CEEMOUT (dispatch a message) and CEE3DMP (generate dump) services are running under CICS, both the messages and dumps are sent to a transient data queue called CESE, and not to their usual destinations. These

usual destinations are the *ddname* specified in the MSGFILE runtime option for messages and the *ddname* given in the *fname* argument of the CEE3DMP service for dumps. CICS ignores both of these *ddnames*.

Using Language Environment Abend-handling

When a CICS application is running under Language Environment, the action taken when a task is scheduled for abnormal termination depends on whether a CICS HANDLE ABEND is active or not active.

When a HANDLE ABEND is active, Language Environment condition handling does not gain control for any abends or program interrupts, and any user-written condition handlers that have been established by CEEHDLR are ignored. Instead, the action defined in the CICS HANDLE ABEND takes place.

When a CICS HANDLE ABEND is not active, Language Environment condition handling **does** gain control for abends and program interrupts if the runtime option TRAP(ON) is specified. Normal Language Environment condition handling is then performed. If TRAP(OFF) is specified, no error handling takes place; the abend proceeds. For details of the actions taken during normal Language Environment condition handling, see the *z/OS: Language Environment Programming Guide*.

User-written Language Environment condition handlers

The runtime option USRHDLR allows you to register a user-written condition handler at the highest level. At a lower level, for example after a subroutine CALL, you can use the CEEHDLR service to register a condition handler for that level. This lower level handler is automatically unregistered on return from the lower level. If desired you can explicitly unregister it by using the CEEHDLU service. For an explanation of stack levels and for details of the USRHDLR runtime option and the CEEHDLR and CEEHDLU services, see the *z/OS: Language Environment Programming Guide*.

If you write your own user-written Language Environment condition handler (other than in COBOL), you can use most CICS commands, provided they are coded with a NOHANDLE, RESP or RESP2 option, to prevent further conditions being raised during execution of the condition handler. The only commands you cannot use are the following, which must not appear in either the condition handler or any program it calls:

- ABEND
- HANDLE ABEND
- HANDLE AID
- HANDLE CONDITION
- IGNORE CONDITION
- POP HANDLE
- PUSH HANDLE

Unless you use the NOLINKAGE translator option (see “NOLINKAGE” on page 80), do not use the CICS translator to translate a COBOL user-written condition handler that you have registered for a routine using the CEEHDLR service. This is because the CICS translator adds two extra arguments to the PROCEDURE DIVISION header of the COBOL program, the EXEC Interface Block (EIB) and the COMMAREA. These arguments do not match the arguments passed by Language Environment. A COBOL condition handler cannot, therefore, contain any CICS commands.

However, a user-written condition handler can call a subroutine to perform CICS commands (and this could be a COBOL routine). If you need to pass arguments to this subroutine, place two dummy arguments before them in the caller. The called subroutine must issue EXEC CICS ADDRESS EIB(DFHEIPTR) before executing any other CICS commands.

For full details of the required interface to any Language Environment condition handling routine, see the *z/OS: Language Environment Programming Guide*.

Managing Language Environment storage

When each program is first used, Language Environment tells CICS how much storage the run unit work area (RUWA) requires. The allocation of storage depends on the setting of the CICS system initialization parameter, RUWAPool.

If you specify RUWAPool=NO, at the start of each CICS link level, CICS issues a GETMAIN for this storage and passes it to Language Environment to use for its control blocks and for storage areas such as STACK, LIBSTACK, and HEAP. The storage is acquired in the default key specified on the transaction. The storage is freed (using FREEMAIN) when the program terminates.

If you specify RUWAPool=YES, the first run of a transaction is the same as with RUWAPool=NO, but CICS keeps a history of the total storage for RUWAs that is requested to run the transaction. This means that when the transaction is run again, CICS issues a single GETMAIN for the total storage (and a single FREEMAIN at task end), creating a RUWAPool. If the transaction follows the same path, CICS allocates the storage from the RUWAPool, and no further GETMAIN has to be issued. If more storage is required for RUWAs because of different or extra CICS links, CICS issues a GETMAIN and updates the history, so that next time the single GETMAIN (and FREEMAIN) is for the larger amount. For transactions that issue a large number of CICS LINK commands, the performance improvement can be considerable.

If you specify AUTODST=YES, CICS indicates to Language Environment that it is able to support dynamic storage tuning (see the *CICS System Definition Guide*).

If a program specifies a runtime option of ALL31(OFF) and Language Environment needs to use storage below the 16MB line, two areas of storage are allocated, one below 16MB and one above the 16MB line.

If necessary, any application can obtain CICS DATAKEY or USER DATAKEY storage by using a CICS GETMAIN command. However, a program with an EXEC KEY of USER cannot use CICS DATAKEY storage.

Mixing languages in Language Environment

This section tells you how to build an application that is composed of programs that have been written in different high-level source languages.

Assembler subroutines called from an HLL program are fairly straightforward and not uncommon. A subroutine called from one HLL but written in another needs much more careful consideration and involves what is called interlanguage communication (ILC). Language Environment defines an ILC application as one built of two or more HLLs and, optionally, assembler. See *z/OS Language Environment Writing Interlanguage Communication Applications* for full details on this subject.

Language Environment dictates that if there is any ILC within a run unit under CICS, each compile unit must be compiled with a Language Environment-conforming compiler. CICS supports three HLLs: C/C++, COBOL, and PL/I. We consider the interfaces in pairs. If your application contains only two HLLs, consult the appropriate section. If your application contains all three HLLs, consult those sections corresponding to each of the interfaces within your application.

C/C++ and COBOL

The conditions under which Language Environment supports ILC between routines written in C/C++ and COBOL depend on the following:

- Whether the language is C or C++
- Which COBOL compiler is being used and whether or not DLL is specified as a compiler option
- Whether the call is static or dynamic
- Whether the function being invoked is within the module or exported from a DLL
- Whether or not the program is reentrant
- What, if any, #pragma linkage statement you have in your C program
- Whether your C program exports functions or variables
- What, if any, extern statement you have in your C++ program

The results of all this are specified in five tables in *z/OS Language Environment Writing Interlanguage Communication Applications*; you should consult this book if your application mixes C/C++ and COBOL.

C/C++ and PL/I

Under CICS, if all the components of your C/C++ and PL/I application are reentrant, Language Environment supports ILC between routines compiled by OS/390 C/C++ and PL/I for MVS & VM or VisualAge PL/I for OS/390 as follows:

- C/C++ routines can statically call PL/I routines and PL/I routines can statically call C/C++ routines.
- C/C++ routines can fetch() PL/I routines that have OPTIONS(FETCHABLE) specified. If the called routine contains any CICS commands, then C/C++ must pass the EIB and the COMMAREA as the first two parameters on the call statement.
- PL/I routines can FETCH only those OS/390 C/C++ routines that have not been processed by the CICS translator. This is because during the dynamic call certain static fields created by the translator cannot be correctly set.

COBOL and PL/I

Under CICS, Language Environment supports ILC between routines compiled with Language Environment-supported COBOL and PL/I compilers, as follows:

- COBOL routines can statically call PL/I routines, and PL/I routines can statically call COBOL routines.
- COBOL programs can dynamically call PL/I routines that have OPTIONS(FETCHABLE) specified and PL/I routines can FETCH COBOL programs.

If the called routine contains any CICS commands then the calling routine must pass the EIB and the COMMAREA as the first two parameters on the CALL statement.

Assembler

The following rules apply:

- You can make static or dynamic calls from any Language Environment-conforming HLL program to a Language Environment-conforming assembler subroutine. Conversely, a Language Environment-conforming assembler routine can make a static call to any Language Environment-conforming routine, and can dynamically load another routine, either assembler or HLL, by using either of the Language Environment macros CEEFETCH or CEELoad.
- You cannot delete (release) an ILC module that has been loaded using CEELoad.
- You can use the CEERELES macro to release an ILC module which has been fetched using CEEFETCH.
- Use the language that fetched it to delete an assembler routine. This can only be done from C/C++, COBOL, and PL/I, if there is no ILC with PL/I in the module being released.

Additionally, you can make static calls from any Language Environment-conforming HLL program or assembler subroutine to a non-conforming assembler subroutine. However, a non-conforming assembler routine cannot make a static call to any Language Environment-conforming routine, nor can it fetch or load a conforming routine, since it cannot use the Language Environment macros.

Note: For assembler to call C or C++, you must include the following statement:

```
C           #pragma linkage(,OS)
C++        extern "OS"
```

DL/I

If you are using DL/I in your ILC application under CICS, calls to DL/I, either by an EXEC DLI statement or by a CALL xxxTDLI, can be made only in programs with the same language as the main program.

Language Environment does not support CALL CEETDLI under CICS.

Using Dynamic Link Libraries and Language Environment

The z/OS dynamic link library (DLL) facility provides a mechanism for packaging programs and data into load modules (DLLs) that can be accessed from other separate load modules. A DLL can export symbols representing routines that can be called from outside the DLL, and can import symbols representing routines or data or both in other DLLs, avoiding the need to link the target routines into the same load module as the referencing routine. When an application references a separate DLL for the first time, the system automatically loads the DLL into memory.

You should define all potential DLL executable modules as PROGRAM resources to CICS.

DLL support is available for applications under CICS where the code has been compiled using any of the compilers listed in the *z/OS Language Environment Programming Guide*. See that manual for more information on building and using DLLs.

Defining runtime options for Language Environment

Language Environment provides runtime options to control your program's processing. Under CICS, exactly which options apply to the execution of a particular program depends not only on the program, but also on how it is run. Java programs and programs initiated from the Web or through CICS IOP services use the Language Environment preinitialization module, CEEPIPI. This has its own version of the CEEDOPT CSECT and such programs get their runtime options from this CSECT. For normal CICS tasks, such as those started from a terminal, use any of the methods listed below to set the Language Environment runtime options. The methods are shown in the order in which they are processed. Each setting could be overridden by a following one. This is, in effect, a reverse order of precedence.

1. In the CEEDOPT CSECT, where the installation default options for CICS are located. This CSECT is generated from the CEECOPT sample and is incorporated into the CEECCICS load module.
2. In the CEEROPT CSECT, where the region-wide default options are located. This CSECT is link-edited into a load module of the same name and placed in a data set in the DFHRPL library concatenation for the CICS job.
3. The user replacable program DFHAPXPO (applies to XPLINK programs only).
4. In the CEEUOPT CSECT, where user-supplied application program-level runtime options are located. This CSECT is linked with the application program itself.
5. In the application source code using the programming language options statements, as follows:
 - In C programs, through the `#pragma runopts` statement in the program source. For example:

```
#pragma runopts(rptstg(on))
```
 - In PL/I programs, through the `PLIXOPT` declaration statement within the program. For example:

```
DECLARE PLIXOPT CHARACTER(18) VARYING STATIC EXTERNAL INIT('RPTOPTS(ON) NOSTAE');
```

Note: There is no source code mechanism that allows the setting of runtime options within COBOL programs or within C++ programs.

6. In the Language Environment options specified in a debugging profile. For more information, see “Debugging profiles” on page 658.

In most installations, the first method in the list above is not available to application programmers, and the second is often not available. However, application programmer can use the last two methods. They are in effect equivalent (some of the newer compilers make them equivalent by generating a CEEUOPT CSECT when `PLIXOPT` is declared). Choose either methods 3 or method 4; do not attempt to use both methods. For details of generating a CEEUOPT CSECT to link with your application, see *z/OS Language Environment Customization*.

Notes:

1. Both CEEDOPT and CEEROPT are able to set any option so that it cannot be overridden by a later specification.
2. Under CICS many of the Language Environment option settings are ignored. These are all the Fortran-only options plus the following:

ABPERC	AIXBLD	CBLOPTS	CBLQDA
DEBUG	EXECOPS	INTERRUPT	LIBRARY
MSGFILE	NONIPTSTACK	PLITASKCOUNT	POSIX

3. CICS provides an environment variable called CICSVAR to allow the CONCURRENCY and API program attributes to be closely associated with the application program itself by using the ENVAR runtime option. Whilst it may be used in a CEEUOPT CSECT to set an installation default, it is most useful to be set in a CEEUOPT CSECT linkedited with an individual program, or set via a #pragma statement in the source of a C or C++ program, or set via a PLIXOPT statement in a PL/I program.

For example, when a program has been coded to threadsafe standards it can be defined as such without having to change an PROGRAM resource definition, or adhere to an installation defined naming standard to allow a program autoinstall exit to install it with the correct attributes. CICSVAR can be used for Language Environment conforming assembler, for PLI, for COBOL and for C and C++ programs (both those compiled with the XPLINK option, and those compiled without it) that have been compiled using a Language Environment conforming compiler. CICSVAR cannot be used for assembler programs that are not Language Environment conforming or for java programs.

Use of CICSVAR overrides the settings on a PROGRAM resource definition installed via standard RDO interfaces, or via program autoinstall. Prior to the program being run for the first time, an INQUIRE PROGRAM command shows the keyword settings from the program definition. Once the application has been run once, an INQUIRE PROGRAM command shows the settings with any CICSVAR overrides applied. CICSVAR can take one of three values, QUASIRENT, THREADSAFE or OPENAPI.

CICSVAR=QUASIRENT results in a program with attributes
CONCURRENCY(QUASIRENT) APIST(CICSAPI)

CICSVAR=THREADSFAE results in a program with attributes
CONCURRENCY(THREADSFAE) APIST(CICSAPI)

CICSVAR=OPENAPI results in a program with attributes
CONCURRENCY(THREADSFAE) APIST(OPENAPI)

An example of ENVAR coded in a CEEUOPT CSECT is given below:

```
CEEUOPT CSECT
CEEUOPT AMODE ANY
CEEUOPT RMODE ANY
CEEUOPT ENVAR=('CICSVAR=THREADSFAE')
END
```

This can be assembled and linkedited into a into a load module and then the CEEUOPT load module linkedited together with any language program supported by Language Environment as explained above.

Alternatively for C and C++ programs, add the following statement at the start of the program source before any other C statements:

```
#pragma runopts(ENVAR(CICSVAR=THREADSFAE))
```

For PL/I programs add the following statement following the PL/I MAIN procedure statement:

```
DCL PLIXOPT CHAR(25) VAR STATIC EXTERNAL INIT('ENVAR(CICSVAR=THREADSFAE)');
```

Finally, be aware that , after all the above changes, there are two exits that are called in the following order and both of these can change some of the options as follows:

1. By setting the CEEAUE_A_OPTION return parameter of the CEEBXITA Language Environment user exit (apart from the LIBRARY, RTLS, STACK, and VERSION options).
2. In the storage tuning user exit, CEECSTX, the options STACK, LIBSTACK, HEAP, ANYHEAP, and BELOWHEAP can be set.

The storage tuning exit, like the CEEROPT CSECT, is region wide, but CEEBXITA is linked into every program. Language Environment calls CEEBXITA the assembler exit because, like CEECSTX, it is invoked before the environment is fully established and must therefore be coded in assembler.

Language Environment supplies a sample source version of CEEBXITA in the SCEESAMP library (it simply returns to its caller for whatever reason it is called). You can use this as it is or modify it for use as the installation default version. However, you can link-edit a specifically tailored version of CEEBXITA with any application program and this version is then used instead of the installation default version. Take great care if you choose this method since CEEBXITA is invoked for up to five different reasons during the course of program execution, and an application-specific version of CEEBXITA must be capable of handling all these invocations.

For more details on both CEEBXITA and CEECSTX see *z/OS Language Environment Customization*.

Runtime options in child enclaves

Under CICS the execution of a CICS LINK command creates what Language Environment calls a Child Enclave. A new environment is initialized and the child enclave gets its runtime options exactly as discussed above. These runtime options are completely independent of those that existed in the creating enclave. Frequent use of EXEC CICS LINK, and the individual setting of many runtime options could affect performance (see also the CICS system initialization parameter RUWAPOL for CICS LINK performance considerations). A static or dynamic call does not incur these overheads. If you need to use CEEUOPT to specify options, specifying only those options that are different from the defaults will improve performance.

Something similar happens when a CICS XCTL command is executed. In this case we do not get a child enclave, but the existing enclave is terminated and then reinitialized with the runtime options determined for the new program. The same performance considerations apply.

CEEBXITA coding

If you write your own version of CEEBXITA, you must write it in assembler. You can use all CICS commands except the ones listed here, provided you specify the NOHANDLE, RESP or RESP2 option, to prevent conditions being raised during the execution of the exit. These are the commands that cannot be used within CEEBXITA, or any routines called by CEEBXITA:

- ABEND
- HANDLE ABEND
- HANDLE AID
- HANDLE CONDITION
- IGNORE CONDITION

- POP HANDLE
- PUSH HANDLE

Determining which options were used

If you want to know which options were in effect when your program ran, specify the option RPTOPTS(ON). When the program ends this produces a list of all the runtime options used. The list is written to the CESE TD queue. The list contains not only the actual settings of the options, but also their origin, that is, whether they are the default for the installation or the region, or whether they were set by the programmer or in one of the exits.

Note: Do not use RPTOPTS(ON) in a production environment - there is significant overhead and it causes a large amount of data to be written to the CESE queue.

For more information about how to specify Language Environment runtime options and also for their meanings, see *z/OS Language Environment Programming Reference*.

Writing a CEEBINT initialization exit for Language Environment

All programs running under Language Environment invoke a subroutine called CEEBINT at program initialization time, just after invocation of the CEEBXITA and CEECSTX exits. The runtime environment is fully operational at this point and Language Environment calls this program the HLL user exit. Language Environment provides a module containing this program in the SCEELKED library (it simply returns to its caller) and this is, therefore, the installation default version. However, you can also write and link-edit a tailored version in to any program to replace the default.

Ordinary Language Environment coding rules apply to CEEBINT, and you can write it in C, C++, PL/I, or Language Environment-conforming assembler. CEEBINT applies to COBOL programs just as any others, but it cannot be written in COBOL or call COBOL programs. If CEEBINT introduces a second HLL to a program, the rules for mixing HLLs described in “Mixing languages in Language Environment” on page 13 apply.

For more information on the High Level Language user exit, CEEBINT, see the *z/OS Language Environment Programming Guide*.

Chapter 4. Programming in COBOL

The *CICS Transaction Server for z/OS Release Guide* lists the COBOL compilers which are supported by CICS Transaction Server for z/OS, Version 3 Release 1, with details of their service status and support in other CICS releases.

All references to COBOL in CICS Transaction Server for z/OS, Version 3 Release 1 documentation imply the use of a supported Language Environment-conforming compiler such as Enterprise COBOL for z/OS, unless specifically stated otherwise. Programs compiled by Language Environment-conforming compilers can take advantage of all Language Environment facilities in a CICS region. The only COBOL compiler which has runtime support in CICS Transaction Server for z/OS, Version 3 Release 1, but is **not** Language Environment-conforming, is the VS COBOL II compiler.

Refer to the *Enterprise COBOL for z/OS: Compiler and Run-Time Migration Guide* for information about migrating between COBOL compilers.

Support for VS COBOL II

In CICS Transaction Server for z/OS, Version 3 Release 1, applications compiled with a VS COBOL II compiler execute using the Language Environment runtime library routines. The native runtime library provided with VS COBOL II is not supported.

“Considerations for VS COBOL II programs” on page 26 lists some restrictions and considerations associated with programs compiled with the VS COBOL II compiler.

In some circumstances, you might need to adjust Language Environment runtime options to enable these applications to execute correctly. The *Enterprise COBOL for z/OS: Compiler and Run-Time Migration Guide* has more information about executing VS COBOL II programs within the Language Environment runtime, and also about converting VS COBOL II programs to Enterprise COBOL.

Support for OS/VS COBOL

In CICS Transaction Server for z/OS, Version 3 Release 1, runtime support for OS/VS COBOL programs is withdrawn. If you attempt to use an OS/VS COBOL program, the abend code ALIK is issued, and CICS abnormally terminates the task and disables the program.

OS/VS COBOL programs must be upgraded to Language Environment-conforming COBOL, and recompiled against a level of COBOL compiler supported by CICS. Enterprise COBOL for z/OS Version 3 is the recommended compiler.

See Appendix B, “Migration for OS/VS COBOL programs,” on page 671 for notes on converting OS/VS COBOL programs to Enterprise COBOL. The *Enterprise COBOL for z/OS: Compiler and Run-Time Migration Guide* has more detailed information about language differences, and describes facilities to help with conversion.

Support for OO COBOL

In CICS Transaction Server for z/OS, Version 3 Release 1, COBOL class definitions and methods (object-oriented COBOL) cannot be used. This restriction includes both Java classes and COBOL classes.

Modules compiled in earlier CICS releases with the OOCOBOL translator option cannot execute in CICS Transaction Server for z/OS, Version 3 Release 1. The OOCOBOL translator option was used for the older SOM-based (System Object Manager-based) OO COBOL, and runtime support for this form of OO COBOL was withdrawn in z/OS V1.2. The newer Java-based OO COBOL, which is used in Enterprise COBOL, is not supported by the CICS translator.

If you have existing SOM-based OO COBOL programs, you should rewrite your OO COBOL into procedural (non-OO) COBOL in order to use the Enterprise COBOL compiler. Note that the newer Java-based OO COBOL is not compatible with SOM-based OO COBOL programs, and is not intended as a migration path for SOM-based OO COBOL programs.

This chapter describes:

- “COBOL programming considerations”
- “Considerations for VS COBOL II programs” on page 26
- “Using based addressing with COBOL” on page 27
- “Calling subprograms from COBOL” on page 28
- “Using the COBOL2 and COBOL3 translator options” on page 34

COBOL programming considerations

Some features of COBOL that are of interest to the CICS programmer are:

- Simplified based addressing using cell pointer variables and the ADDRESS special register.

- The ability to use COBOL CALL statements to call assembler language, other COBOL programs, and PL/I or C/C++ programs.
- The LENGTH special register, which CICS uses to deduce the length of data items.
- The ability to use the RETURN-CODE special register in a CICS application program. This register allows you to set and access return codes in COBOL programs.

With compiler option DATA(24), the WORKING-STORAGE SECTION is allocated below the 16MB line. With compiler option DATA(31), the WORKING-STORAGE SECTION is allocated above the 16MB line.

COBOL programming restrictions

This section describes COBOL language elements that you cannot use under CICS, or whose use is restricted or can cause problems under CICS.

By default, the CICS translator and the COBOL compiler do not detect the use of COBOL words affected by the restrictions listed here. The use of a restricted word in a CICS environment may cause a failure at execution time. However, COBOL provides a reserved-word table, IGYCCICS, for CICS application programs. If you specify the compiler option WORD(CICS), the compiler uses IGYCCICS, and COBOL words that are not supported under CICS are flagged by the compiler with an error message. (The COBOL words normally restricted by the default IBM-supplied reserved-word table are also flagged.) See the *Enterprise COBOL for z/OS: Programming Guide* for a current listing of the words which are restricted by IGYCCICS.

The following restrictions apply to a COBOL program that is to be used as a CICS application program. (See the appropriate COBOL programming guide for more information about these functions.)

- If no IDENTIFICATION DIVISION is present, only the CICS commands are expanded. If the IDENTIFICATION DIVISION only is present, only DFHEIVAR, DFHEIBLK, and DFHCOMMAREA are produced.
- Statements that produce variable-length areas, such as OCCURS DEPENDING ON, should be used with caution within the WORKING-STORAGE SECTION.
- If you are running CICS applications written in COBOL under Language Environment for the first time, you may need to review the Language Environment runtime options in use at your installation. In particular, if your applications are not coded to ensure that the WORKING-STORAGE SECTION is properly initialized (for example, cleared with binary zeros before sending maps), you should use the STORAGE runtime option. See *z/OS Language Environment Customization* for information about customizing Language Environment runtime options.
- You cannot use entry points in COBOL in CICS.
- When a debugging line is to be used as a comment, it must not contain any unmatched quotation marks.
- Do not use EXEC CICS commands in a Declaratives Section.
- You must use CICS commands for most input and output processing. Therefore, do not describe files or code any OPEN, CLOSE, READ, START, REWRITE, WRITE, or DELETE statements. Instead, use CICS commands to retrieve, update, insert, and delete data.

- Do not use a format-1 ACCEPT statement in a CICS program. Format-2 ACCEPT statements are supported by Language Environment enabled compilers.
 - Do not use DISPLAY . . . UPON CONSOLE and DISPLAY . . . UPON SYSPUNCH. DISPLAY to the system logical output device (SYSOUT, SYSLIST,SYSLST) is supported.
 - Do not use STOP “literal”.
 - There are restrictions on the use of the SORT statement. See the *Enterprise COBOL for z/OS: Programming Guide* for information. Do not use MERGE.
 - Do not use:
 - USE declaratives.
 - ENVIRONMENT DIVISION and FILE SECTION entries associated with data management, because CICS handles data management. (These can be used when they are associated with the limited SORT facility referenced above.)
 - User-specified parameters to the main program.
 - Do not use the following compiler options:
 - DYNAM (if program is to be translated)
 - NOLIB (if program is to be translated)
 - NORENT
- You may use the DLL compiler option.
- The use of the TEST(SYM,NOSEPARATE) compiler option results in a very large increase in program size. Therefore, short-of-storage problems may occur when using this option. You can achieve the same functionality with TEST(SYM,SEPARATE) without an increase in program size. For more information about the TEST compiler option, see the *Enterprise COBOL for z/OS: Programming Guide*.
 - Use TRUNC(OPT) for handling binary data items if they conform to the PICTURE definitions. Otherwise use TRUNC(OPT) as the compiler option, and USAGE COMP-5 for items where the binary value might be larger than the PICTURE clause would allow. TRUNC(BIN) inhibits runtime performance, so only use this option if you have no control over binary data items (such as those created by a code generator). (TRUNC(STD) is the default.)
 - For VS COBOL II programs with Language Environment runtime, the following limits apply to the length of WORKING-STORAGE:
 - When the compiler option DATA(24) is used, the limit is the available space below the 16MB line.
 - When the compiler option DATA(31) is used, the limit is 128MB.

80 bytes are required for storage accounting and save areas, and this must be included within the limits.
 - The use of the RMODE(24) compiler option means that the program always resides below the 16MB line, so this is not recommended. RMODE(ANY) or RMODE(AUTO) should be used instead. For more information about the RMODE compiler option, see the *Enterprise COBOL for z/OS: Programming Guide*.
 - If the DLI option is specified and an ENTRY statement immediately follows the PROCEDURE DIVISION header, you are recommended to change the PROGRAM-ID name to the ENTRY statement literal, then delete the ENTRY statement.
 - The following compiler options have no effect in a CICS environment:
 - ADV
 - AWO
 - EXPORTALL


```

FASTSRT
NAME
OOCOBOL
OUTDD
THREAD

```

- If you use HANDLE CONDITION or HANDLE AID, you can avoid addressing problems by using SET(ADDRESS OF A-DATA) or SET(A-POINTER) where A-DATA is a structure in the LINKAGE SECTION and A-POINTER is defined with the USAGE IS POINTER clause.
- For a COBOL program running above the 16MB line, these restrictions apply for 31-bit addressing:
 - If the receiving program is link-edited with AMODE(31), addresses passed to it must be 31-bits long (or 24-bits long with the left-most byte set to zeros).
 - If the receiving program is link-edited with AMODE(24), addresses passed to it must be 24-bits long.

Specify the DATA(24) compiler option for programs running in 31-bit addressing mode that are passing data arguments to programs in 24-bit addressing mode. This ensures that the data will be addressable by the called program.

Language Environment CBLPSHPOP option

The CBLPSHPOP runtime option controls whether Language Environment automatically issues an EXEC CICS PUSH HANDLE command during initialization and an EXEC CICS POP HANDLE command during termination whenever a COBOL subroutine is called.

If your application makes many COBOL subroutine CALLs under CICS, performance is better with CBLPSHPOP(OFF) than with CBLPSHPOP(ON). You can set CBLPSHPOP on an individual transaction basis by using CEEUOPT, as explained in “Defining runtime options for Language Environment” on page 16. However, because condition handling has not been stacked, be aware that:

- If your called routine raises a condition that causes CICS to attempt to pass control to a condition handler in the calling routine, this is an error and your transaction will be abnormally terminated.
- If you use any of the PUSHable CICS commands, HANDLE ABEND, HANDLE AID, HANDLE CONDITION, or IGNORE CONDITION, within the called routine, you will be changing the settings of your caller and this could lead to later errors.

Using the DL/I CALL interface

If you have programs that use CALL DL/I, and you have not yet made the following changes to them, you should now do so:

- Retain the user interface block (DLIUIB) declaration and at least one program control block (PCB) declaration in the LINKAGE SECTION.
- Change the PCB call to specify the UIB directly, as follows:

```

CALL 'CBLTDLI' USING PCB-CALL
                      PSB-NAME
                      ADDRESS OF DLIUIB.

```

- Obtain the address of the required PCB from the address list in the UIB.

Figure 1 on page 26 illustrates the whole of the above process. The example in the figure assumes that you have three PCBs defined in the PSB and want to use the second PCB in the database CALL. Therefore, when setting up the ADDRESS special register of the LINKAGE SECTION group item PCB, the program uses 2 to index the working-storage table, PCB-ADDRESS-LIST. To use the nth PCB, you

use the number n to index PCB-ADDRESS-LIST.

```

WORKING-STORAGE SECTION.
  77 PCB-CALL          PIC X(4) VALUE 'PCB '.
  77 GET-HOLD-UNIQUE   PIC X(4) VALUE 'GHU '.
  77 PSB-NAME          PIC X(8) VALUE 'CBLPSB'.
  77 SSA1              PIC X(40) VALUE SPACES.
  01 DLI-IO-AREA.
    02 DLI-IO-AREA1    PIC X(99).
*
LINKAGE SECTION.
  COPY DLIUIB.
  01 OVERLAY-DLIUIB REDEFINES DLIUIB.
    02 PCBADDR          USAGE IS POINTER.
    02 FILLER           PIC XX.
  01 PCB-ADDR-LIST.
    02 PCB-ADDRESS-LIST USAGE IS POINTER
                          OCCURS 10 TIMES.
  01 PCB.
    02 PCB-DBD-NAME     PIC X(8).
    02 PCB-SEG-LEVEL    PIC XX.
    02 PCB-STATUS-CODE  PIC XX.
*
PROCEDURE DIVISION.
*SCHEDULE THE PSB AND ADDRESS THE UIB
  CALL 'CBLTDLI' USING PCB-CALL PSB-NAME ADDRESS OF DLIUIB.
*
*MOVE VALUE OF UIBPCBAL, ADDRESS OF PCB ADDRESS LIST (HELD IN UIB)
*(REDEFINED AS PCBADDR, A POINTER VARIABLE), TO
*ADDRESS SPECIAL REGISTER OF PCB-ADDR-LIST TO PCBADDR.
  SET ADDRESS OF PCB-ADDR-LIST TO PCBADDR.
*MOVE VALUE OF SECOND ITEM IN PCB-ADDRESS-LIST TO ADDRESS SPECIAL
*REGISTER OF PCB, DEFINED IN LINKAGE SECTION.
  SET ADDRESS OF PCB TO PCB-ADDRESS-LIST(2).
*PERFORM DATABASE CALLS .....
  .....
  MOVE ..... TO SSA1.
  CALL 'CBLTDLI' USING GET-HOLD-UNIQUE PCB DLI-IO-AREA SSA1.
*CHECK SUCCESS OF CALLS .....
  IF UIBFCTR IS NOT EQUAL LOW-VALUES THEN
                                  ..... error diagnostic code
  .....
  IF PCB-STATUS-CODE IS NOT EQUAL SPACES THEN
                                  ..... error diagnostic code
  .....

```

Figure 1. Using the DL/I CALL interface

Considerations for VS COBOL II programs

Language Environment provides support for the execution of programs compiled by the VS COBOL II compiler. The native runtime library for this compiler is not supported. However, this compiler is not Language Environment-conforming (it is a pre-Language Environment compiler), so there are some restrictions and considerations associated with its use.

For detailed information on migrating VS COBOL II programs to Language Environment support, see the *Enterprise COBOL for z/OS: Compiler and Run-Time Migration Guide*.

Language Environment callable services

Programs compiled by Language Environment-conforming COBOL compilers can use all Language Environment callable services, either dynamically or statically. However, for CICS applications, the CEEMOUT (dispatch a message)

and CEE3DMP (generate dump) services differ, in that the messages and dumps are sent to the CESE transient data queue rather than to the ddname specified in the MSGFILE runtime option.

VS COBOL II programs can make dynamic calls to the date and time callable services, but no other calls, either static or dynamic, to Language Environment callable services are supported for VS COBOL II programs.

Re-linking VS COBOL II programs

If object modules are not available for re-linking existing VS COBOL II programs to use the runtime support provided by Language Environment, a sample job stream for performing the task is provided in the IGZWRLKA member of the SCEESAMP sample library.

CICS stub

Although COBOL programs linked with the old CICS stub, DFHECI, run under Language Environment, use of the DFHELII stub is recommended and is essential in a mixed language environment. DFHECI has to be link-edited at the top of your application, but DFHELII can be linked anywhere in the application.

Using CEEWUCHA

If you are adapting VS COBOL II programs to use the runtime support provided by Language Environment, the sample user condition handler, CEEWUCHA, supplied by Language Environment in the SCEESAMP library, can be used to advantage. It does the following:

- It provides compatibility with existing VS COBOL II applications running under CICS by allowing EXEC CICS HANDLE ABEND LABEL statements to get control when a runtime detected error occurs.
- It converts all unhandled runtime detected errors to the corresponding user 1xxx abend issued by VS COBOL II.
- It suppresses all IGZ0014W messages, which are generated when IGZETUN or IGZEOPT is link-edited with a VS COBOL II application. (Performance is better if the programs are not link-edited with IGZETUN or IGZEOPT.)

Using based addressing with COBOL

CICS application programs need to access data dynamically when the data is in a CICS internal area, and only the address is passed to the program. Examples are:

- CICS areas such as the CWA, TWA, and TCTTE user area (TCTUA), accessed using the ADDRESS command
- Input data, obtained by EXEC CICS commands such as READ and RECEIVE with the SET option

COBOL provides a simple method of obtaining addressability to the data areas defined in the LINKAGE SECTION using pointer variables and the ADDRESS special register. Figure 2 on page 28 gives an example of this.

The ADDRESS special register holds the address of a record defined in the LINKAGE SECTION with level 01 or 77. This register can be used in the SET option of any command in ADDRESS mode. These commands include GETMAIN, LOAD, READ, and READQ. For programming information, including a complete list of these commands, see the *CICS Application Programming Reference* manual..

Figure 2 on page 28 shows the use of ADDRESS special registers in COBOL. If the records in the READ or REWRITE commands are of fixed length, no LENGTH option is required. This example assumes variable-length records. After the read, you can get the length of the record from the field named in the LENGTH option

(here, LRECL-REC1). In the REWRITE command, you must code a LENGTH option if you want to replace the updated record with a record of a different length.

```
WORKING-STORAGE SECTION.  
77 LRECL-REC1    PIC S9(4) COMP.  
LINKAGE SECTION.  
01 REC-1.  
    02 FLAG1      PIC X.  
    02 MAIN-DATA  PIC X(5000).  
    02 OPTL-DATA  PIC X(1000).  
01 REC-2.  
    02 ...  
PROCEDURE DIVISION.  
    EXEC CICS READ UPDATE...  
        SET(ADDRESS OF REC-1)  
        LENGTH(LRECL-REC1)  
        END-EXEC.  
    IF FLAG1 EQUAL X'Y'  
        MOVE OPTL-DATA TO ...  
        :  
        :  
    EXEC CICS REWRITE...  
        FROM(REC-1)  
        END-EXEC.
```

Figure 2. Addressing CICS data areas in locate mode

Using WITH DEBUGGING MODE

If a “D” is placed in column seven of the first line of a COBOL EXEC CICS command, that “D” is also found in the translated CALL statements. This translated command is only executed if WITH DEBUGGING MODE is specified. A “D” placed on any line other than the first line of the EXEC CICS statement is not required and is ignored by the translator.

Calling subprograms from COBOL

In a CICS system, when control is transferred from the active program to an external program, but the transferring program remains active and control can be returned to it, the program to which control is transferred is called a subprogram.

There are three ways of transferring control to a subprogram:

EXEC CICS LINK

The calling program contains a command in one of these forms:

```
EXEC CICS LINK PROGRAM('subpgname')  
EXEC CICS LINK PROGRAM(name)
```

In the first form, the called subprogram is specified as an alphanumeric literal. In the second form, name refers to the COBOL data area with length equal to that required for the name of the subprogram.

Static COBOL call

The calling program contains a COBOL statement of the form:

```
CALL 'subpgname'
```

The called subprogram is explicitly named as a literal string.

Dynamic COBOL call

The calling program contains a COBOL statement of the form:

```
CALL identifier
```

The identifier is the name of a COBOL data area that must contain the name of the called subprogram.

For information about the performance implications of using each of these methods to call a subprogram, see the *Enterprise COBOL for z/OS: Programming Guide*, and the *IBM Enterprise COBOL Version 3 Release 1 Performance Tuning Paper*. The White Paper is available on the Web at www.ibm.com/software/ad/cobol/library

COBOL programs can call any language programs statically or dynamically. LINK or XCTL are not required for inter-language communication, unless you wish to use CICS functions such as COMMAREA. See *Language Environment Writing Inter Language Communication Applications* for guidance on mixing languages under Language Environment control.

“Rules for calling subprograms” on page 30 gives the rules governing the use of the three ways to call a subprogram. This information refers to CICS application logical levels. Each LINK command creates a new logical level, the called program being at a level one lower than the level of the calling program (CICS is taken to be at level 0). Figure 3 on page 33 shows logical levels and the effect of RETURN commands and CALL statements in linked and called programs.

The term run unit is used in Figure 3 on page 33. A run unit is a running set of one or more programs that communicate with each other by COBOL static or dynamic CALL statements. In a CICS environment, a run unit is entered at the start of a CICS task, or invoked by a LINK or XCTL command. A run unit can be defined as the execution of a program defined by a PROGRAM resource definition, even though for dynamic CALL, the subsequent PROGRAM definition is needed for the called program. When control is passed by a XCTL command, the program receiving control **cannot** return control to the calling program by a RETURN command or a GOBACK statement, and is therefore not a subprogram.

The terms 'translator' and 'translation' in “Rules for calling subprograms” on page 30 refer to the separate translator. This step is not required if a compiler with an integrated translator is used.

Rules for calling subprograms

The following rules describe the requirements and behavior of called or linked subprograms.

Translation

LINK

The linked subprogram must be translated if it, or any subprogram invoked from it, contains CICS function.

Static and Dynamic COBOL CALL

The called subprogram must be translated if it contains CICS commands or references to the EXEC interface block (DFHEIBLK) or to the CICS communication area (DFHCOMMAREA).

Compilation

You **must** always use the NODYNAM compiler option (the default) when you compile a COBOL program that is to run with CICS, even if the program issues dynamic calls.

Link-editing

LINK

The linked subprogram must be compiled and link-edited as a separate program.

Static COBOL CALL

The called subprogram must be link-edited with the calling program to form a single load module (but the programs can be compiled separately). This can produce large program modules, and it also stops two programs that call the same program from sharing a copy of that program.

Dynamic COBOL CALL

The called subprogram must be compiled and link-edited as a separate load module. It can reside in the link pack area or in a library that is shared with other CICS and non-CICS regions at the same time.

CICS CSD entries without program autoinstall

(If you use program autoinstall, you do not need an entry in the CSD.)

LINK

The linked subprogram must be defined using RDO. If the linked subprogram is unknown or unavailable, even though autoinstall is active, the LINK fails with the PGMIDERR condition.

Static COBOL CALL

The calling program must be defined in the CSD. If program A calls program B and then program B attempts to call program A, COBOL issues a message and an abend(1015). The subprogram is part of the calling program so no CSD entry is required.

Dynamic COBOL CALL

The calling program must be defined in the CSD. If program A calls program B and then program B attempts to call program A, COBOL issues a message and an abend(1015). The called subprogram must be defined in the CSD. If the called subprogram cannot be loaded or is unavailable even though autoinstall is active, COBOL issues a message and abends (1029).

Return from subprogram

LINK

The linked subprogram must return using either RETURN or a native language return command such as the COBOL statement GOBACK.

Static and Dynamic COBOL CALL

The called subprogram must return using a native language return statement such as the COBOL statement GOBACK or EXIT PROGRAM. The use of RETURN in the called subprogram terminates the calling program.

Language of subprogram

LINK, Static and Dynamic COBOL CALL

Any language supported by CICS.

Contents of subprogram

The contents of any called or linked subprogram can be any function supported by CICS for the language (including calls to external databases, for example, DB2® and DL/I) with the exception that an assembler language subprogram cannot CALL a lower level subprogram.

Passing parameters to subprogram

Data can be passed by any of the standard CICS methods (COMMAREA, TWA, TCTUA, TS queues) if the called or linked subprogram is processed by the CICS translator.

LINK

If the COMMAREA is used, its address must be passed in the LINK command. If the linked subprogram uses 24-bit addressing, and the COMMAREA is above the 16MB line, CICS copies it to below the 16MB line, and recopies it on return.

Static COBOL CALL

The CALL statement may pass DFHEIBLK and DFHCOMMAREA as the first two parameters, if the called program is to issue EXEC CICS requests, or the called program can issue EXEC CICS ADDRESS commands. The COMMAREA is optional but if other parameters are passed, a dummy COMMAREA must also be passed. The rules for nested programs can be different. See “Nesting: what the application programmer must do” on page 39

Dynamic COBOL CALL

The CALL statement may pass DFHEIBLK and DFHCOMMAREA as the first two parameters, if the called program is to issue EXEC CICS requests, or the called program can issue EXEC CICS ADDRESS commands. The COMMAREA is optional but if other parameters are passed, a dummy COMMAREA must also be passed. If the called subprogram uses 24-bit addressing and any parameter is above the 16MB line, COBOL issues a message and abends(1033) .

Storage

LINK

On each entry to the linked subprogram, a new initialized copy of its WORKING-STORAGE SECTION is provided, and the run unit is reinitialized (in some circumstances, this can cause a performance degradation).

Static and Dynamic COBOL CALL

On the first entry to the called subprogram within a CICS logical level, a new initialized copy of its WORKING-STORAGE SECTION is provided. On subsequent entries to the called subprogram at the same logical level, the same WORKING STORAGE is provided in its last-used state, that is, no storage is freed, acquired, or initialized. If performance is unsatisfactory with LINK commands, COBOL calls may give improved results.

CICS condition, AID and abend handling

LINK

On entry to the called subprogram, no abend or condition handling is active. Within the subprogram, the normal CICS rules apply. In order to establish an abend or condition handling environment, that exists for the duration of the subprogram, a new HANDLE command should be issued on entry to the subprogram. The environment so created remains in effect until either a further HANDLE command is issued, or the subprogram returns control to the caller.

Static and Dynamic COBOL CALL

If the dynamic COBOL CALL fails, CICS abend handling is not invoked, and you may get a COBOL abend code (1013). If the dynamic COBOL CALL fails, with Language Environment and CBLPSHPOP ON:

- On entry to the called subprogram, no abend or condition handling is active. Within the subprogram, the normal CICS rules apply. On entry to the called subprogram, COBOL issues a PUSH HANDLE to stack the calling program's condition or abend handlers. In order to establish an abend or condition handling environment that exists for the duration of the subprogram, a new HANDLE command should be issued on entry to the subprogram. The environment that this creates remains in effect until either a further HANDLE command is issued or the subprogram returns control to the caller. When control is returned to the calling program from the subprogram, COBOL unstacks the condition and abend handlers using a POP HANDLE.

If the dynamic COBOL CALL fails, with CBLPSHPOP OFF:

- The condition/AID and abend handling for the calling program remain in effect.

Location of subprogram

LINK

Can be remote.

Static and Dynamic COBOL CALL

Must be local.

Flow of control between programs and subprograms

Figure 3 on page 33 shows the possible flows between COBOL main and subprograms (also known as *run units*).

A main, or *level 1* program can use the COBOL GOBACK or STOP RUN statements, or the CICS RETURN command to terminate and return to CICS. It can use a COBOL CALL statement to call a subprogram at the same logical level (level 1), or a CICS LINK command to call a subprogram at a lower logical level. A called subprogram at level 1 can return to the caller using the COBOL GOBACK statement, or can terminate and return to CICS using EXEC CICS RETURN.

A subprogram executing at level 2 can use the COBOL GOBACK or STOP RUN statements, or the CICS RETURN command to terminate and return to the level 1 calling program. It can use a COBOL CALL statement or a CICS XCTL command to call a subprogram at the same level (level 2). A subprogram called using the COBOL CALL at level 2 can return to the caller (at level 2) using the COBOL GOBACK statement, or can return to the level 1 calling program using EXEC CICS RETURN. A subprogram called using XCTL at level 2 can only return to the level 1 calling program, using GOBACK, STOP RUN or EXEC CICS RETURN.

See “Application program logical levels” on page 566 for more information about program logical levels.

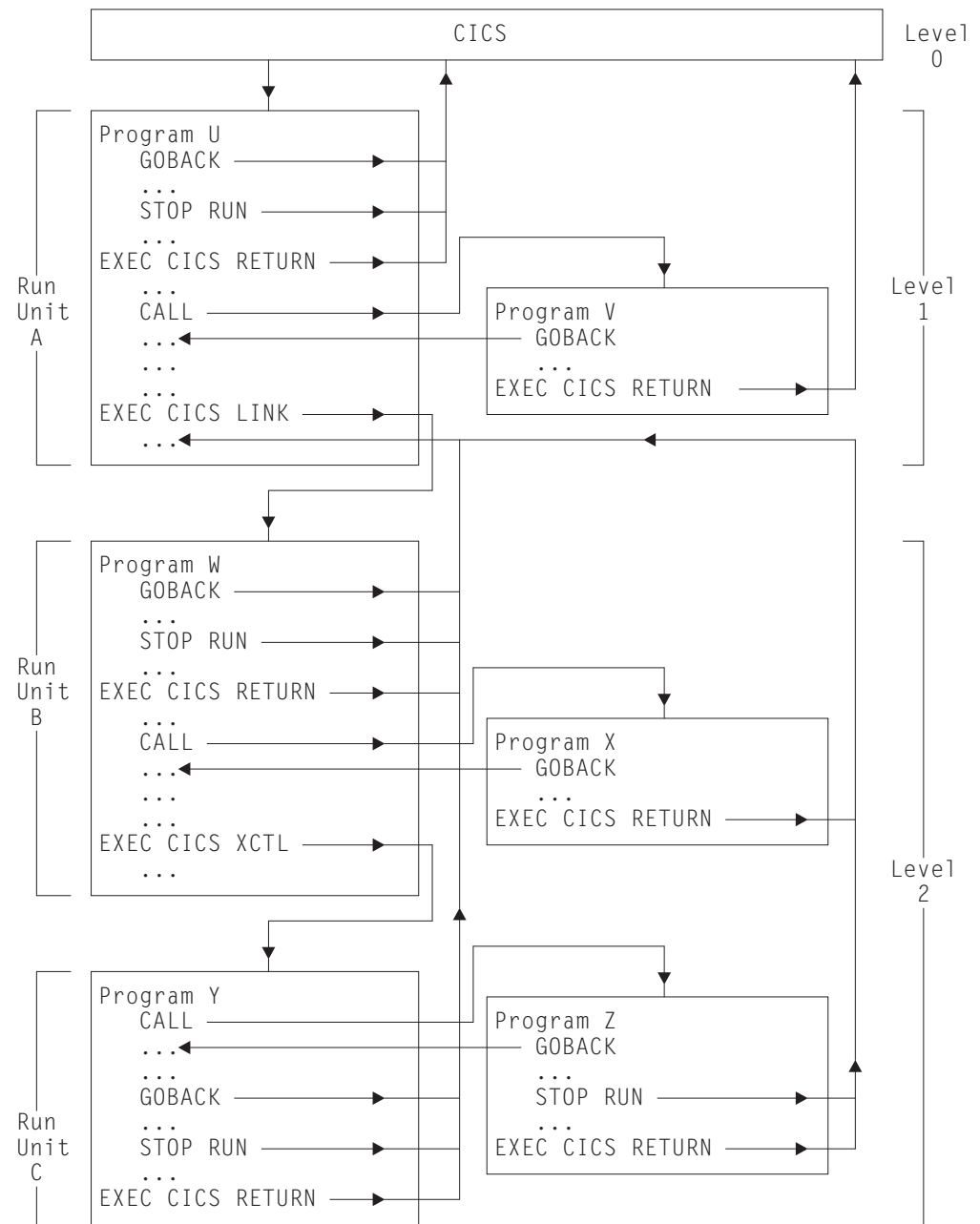


Figure 3. Flow of control between COBOL programs, run units, and CICS

Using the COBOL2 and COBOL3 translator options

For general information about translating your program and preparing it for execution, see Chapter 8, “Translation and compilation,” on page 67.

The following CICS translator options for COBOL are available in CICS Transaction
Server for z/OS, Version 3 Release 1:

• COBOL2
• COBOL3

The ANSI85 translator option ceased to be available in CICS Transaction Server for z/OS, Version 2 Release 2.

Modules compiled in earlier CICS releases with the OOCOBOL translator option cannot execute in CICS Transaction Server for z/OS, Version 3 Release 1. The OOCOBOL translator option was used for the older SOM-based (System Object Manager-based) OO COBOL, and runtime support for this form of OO COBOL was withdrawn in z/OS V1.2. The newer Java-based OO COBOL, which is used in Enterprise COBOL, is not supported by the CICS translator.

The COBOL2 option is the default. It does not have the same effect on the translator as it did in CICS Transaction Server for z/OS, Version 2 Release 1 and earlier releases. COBOL2 instructs the translator to translate as COBOL3, but in addition to include declarations of temporary variables for use in EXEC CICS and EXEC DLI requests.

Choose the COBOL2 option if you are re-translating old programs which were written in such a way that they require the use of temporary variables. In particular, note that the use of temporary variables might circumvent errors that would normally occur when an argument value in a program is incorrectly defined. The COBOL2 option in CICS Transaction Server for z/OS, Version 2 Release 1 and earlier releases provided declarations of temporary variables. Because of this feature, incorrect definitions of argument values might be present, but not noticeable at runtime, in programs that were originally translated with the COBOL2 option in earlier releases of CICS Transaction Server. Translating these programs with the COBOL3 option can reveal these errors for the first time. To assist with migration to the newer releases of CICS, you may use the new COBOL2 option to continue to circumvent the errors in the programs, rather than correcting them.

If you are confident that your program do not need the translator's temporary variables, you may use COBOL3, which results in smaller working storage. The COBOL3 option includes all features of the older COBOL2 and ANSI85 translator options, except for declarations of temporary variables.

The CICS translator support in CICS Transaction Server for z/OS, Version 2
Release 2 and later versions and releases does not support the use of the CMPR2
compiler option previously available with old COBOL compilers. For information on
upgrading these COBOL programs to the NOCMPR2 feature, see the *Enterprise*
COBOL for z/OS: Compiler and Run-Time Migration Guide.

| **Note:** COBOL2 and COBOL3 are mutually exclusive. If you specify both options by
| different methods, the COBOL3 option is *always* used, regardless of where
| the two options have been specified. If this happens, the translator issues a
| warning message.

The following topics describe specific translator action that is taken when the COBOL3 option is used. Processing with the COBOL2 option is the same in all respects, except for declarations of temporary variables.

Literals intervening in blank lines

Blank lines can appear anywhere in a COBOL source program. A blank line
contains nothing but spaces between columns 7 and 72 inclusive.

If blank lines occur within literals in a COBOL source program, the translator eliminates them from the translated output but includes them in the translated listing.

Lower case characters

Lower case characters can occur anywhere in any COBOL word, including user-defined names, system names, and reserved words.

The translator listing and output preserve the case of COBOL text as entered.

In addition, the translator accepts mixed case in:

- Translator options
- EXEC CICS commands, both for keywords and for arguments to keywords
- CBL and PROCESS statements
- Compiler directives such as EJECT and SKIP1

The translator does not translate lower case text into upper case. Some names in COBOL text, for example file names and transaction IDs, must match with externally defined names. Such names must always be entered in the same case as the external definition.

If you specify the LINKAGE translator option, or allow it to default, a mixed-case version of the EIB structure (DFHEIBLC) is inserted into the LINKAGE SECTION.

Sequence numbers containing any character

In a COBOL source program, the sequence number field can contain any character in the computer's character set. The sequence number fields need not be in any order and need not be unique.

REPLACE statement

COBOL programs can include the REPLACE statement, which allows the replacement of identified text by defined substitution text. The text to be replaced and inserted can be pseudo-text, an identifier, a literal, or a COBOL word. REPLACE statements are processed after COPY statements.

If you process your COBOL source statements with the CICS-supplied translator, the translator accepts REPLACE statements but **does not** translate text between pseudo-text delimiters, with the exception of CICS built-in functions (DFHRESP and DFHVALUE), which are translated wherever they occur. CICS commands should not be placed between pseudo-text delimiters.

If you use the integrated translator, the translator accepts REPLACE statements and does translate text between pseudo-text delimiters. CICS commands can be placed between pseudo-text delimiters.

Batch compilation

Separate COBOL programs can be compiled together as one input file. An END PROGRAM header statement terminates each program and is optional for the last program in the batch.

The translator accepts separate COBOL programs in a single input file, and interprets END PROGRAM header statements.

Translator options specified as parameters when invoking the translator are in effect for the whole batch, but can be overridden for a unit of compilation by options specified in the CBL or PROCESS card that initiates the unit.

The options for a unit of compilation are determined according to the following order of priority:

1. Options fixed as installation non-user-modifiable options.
2. Options specified in the CBL or PROCESS card that initiates the unit.
3. Options specified when the translator is invoked.
4. Default options.

For more information about compilation, see Chapter 9, “Installing application programs,” on page 99.

If you are using batch compilation, you must take some additional action to ensure that compilation and linkage editing are successful, as follows:

- Include the compiler NAME option as a parameter in the JCL statement that invokes the compiler or in a CBL statement for each top-level (non-nested) program. This causes the inclusion of a NAME statement at the end of each program. See Figure 4 on page 37 for more information.
- Edit the compiler output to add INCLUDE and ORDER statements for the CICS COBOL stub to each object module. These statements cause the linkage editor to include the stub at the start of each load module. These statements can be anywhere in the module, though by convention they are at the start. You may find it convenient to place them at the end of the module, immediately before each NAME statement. Figure 5 on page 37 shows the output from Figure 4 after editing in this way.

For batch compilation you must vary the procedure described in the Chapter 9, “Installing application programs,” on page 99. The following is a suggested method:

1. Split the supplied cataloged procedure DFHYITVL into two procedures; PROC1 containing the translate and compilation steps (TRN and COB), and PROC2 containing the linkage editor steps COPYLINK and LKED.
2. In PROC1, add the NAME option to the parameters in the EXEC statement for the compiler, which then looks like this:

```
//COB      EXEC PGM=IGYCRCTL,REGION=...,
//          PARM='....,NAME,....',
```
3. In PROC1, change the name and disposition of the compiler output data set &&LOADSET. At least remove the initial && from the data set name and change the disposition to CATLG. The SYSLIN statement should then read:

```
//SYSLIN DD DSN=LOADSET,DISP=(NEW,CATLG),
//          UNIT=&WORK,SPACE=(80,(250,100))
```
4. Run PROC1.

```

.....
....program a....
.....
NAME PROGA(R)
.....
....program b....
.....
NAME PROGB(R)
.....
....program c....
.....
NAME PROGC(R)

```

Figure 4. Compiler output before editing

5. Edit the compiler output in the data set LOADSET to add the INCLUDE and ORDER statements as shown in Figure 5. If you use large numbers of programs in batches, you should write a simple program or REXX EXEC to insert the ORDER and INCLUDE statements.
6. In PROC2, add a DD statement for the library that includes the CICS stub. The standard name of this library is CICSTS31.CICS.SDFHLOAD. The INCLUDE statement for the stub refers to this library by the DD name. In Figure 5, it is assumed you have used the DD name SYSLIB (or concatenated this library to SYSLIB). The suggested statement is:

```

//SYSLIB DD DSN=CICSTS31.CICS.SDFHLOAD,
//        DISP=SHR

```

7. In PROC2, replace the SYSLIN concatenation with the single statement:

```

//SYSLIN DD DSN=LOADSET,
//        DISP=(OLD,DELETE)

```

In this statement it is assumed that you have renamed the compiler output data set LOADSET.

8. Run PROC2.

```

....program a....
.....
INCLUDE SYSLIB(DFHELII)
ORDER DFHELII
NAME PROGA(R)
.....
....program b....
.....
INCLUDE SYSLIB(DFHELII)
ORDER DFHELII
NAME PROGB(R)
.....
....program c....
.....
INCLUDE SYSLIB(DFHELII)
ORDER DFHELII
NAME PROGC(R)

```

Figure 5. Linkage editor input

Note: You are recommended to use the DFHELII stub, but DFHECI is still supplied, and can be used.

Nested programs

- COBOL programs can contain COBOL programs.
- Contained programs are included immediately before the END PROGRAM statement of the containing program.
- A contained program can also be a containing program, that is, it can itself contain other programs.
- Each contained or containing program is terminated by an END PROGRAM statement.

For an explanation of valid calls to nested programs and of the COMMON attribute of a nested program, see the *Enterprise COBOL for z/OS Customization Guide*.

Integrated CICS translator

When using the integrated CICS translator no action is necessary for nested programs that contain EXEC CICS commands. The compiler, in effect, declares DFHEIBLK and DFHCOMMAREA as global in the top-level program. This means that explicit coding of EIB and COMMAREA on the USING phrases on CALL and on the PROCEDURE DIVISION are not required, as described in “Translator action,” for the separate translator.

Existing or modified programs will require appropriate modification for use with the CICS integrated translator. If modification is difficult the user should continue to use separate translation and compilation.

Translator action

The CICS translator treats top-level and nested programs differently.

The translator translates a top-level program (a program that is not contained by any other program) in the normal way, with one addition. The translator assigns the GLOBAL attribute for all translator-generated variables in the WORKING-STORAGE SECTION.

The translator translates nested or contained programs in a special way as follows:

- A DATA DIVISION and LINKAGE SECTION are added if they do not already exist.
- Declarations for DFHEIBLK (EXEC interface block) and DFHCOMMAREA (communication area) are inserted into the LINKAGE SECTION.
- EXEC CICS commands and CICS built-in functions are translated.
- The PROCEDURE DIVISION header is **not** modified.
- No translator-generated temporary variables, used for pre-call assignments, are inserted in the WORKING-STORAGE SECTION.

The translator interprets that the input source starts with a top-level program if the first non-comment record is any of the following:

- IDENTIFICATION DIVISION statement
- CBL card
- PROCESS card

If the first record is none of these, the translator treats the input as part of the PROCEDURE DIVISION of a nested program. The first CBL or PROCESS card indicates the start of a top-level program and of a new unit of compilation. Any IDENTIFICATION DIVISION statements that are found before the first top-level program indicate the start of a new nested program.

The practical effect of these rules is that nested programs cannot be held in separate files and translated separately. A top-level program and all its directly and indirectly contained programs constitute a single unit of compilation and should be submitted together to the translator.

Comments in translator input

The translator treats comments that follow an END PROGRAM statement as belonging to the *next* program in the input source. Comments that precede an IDENTIFICATION DIVISION statement appear in the listing *after* the IDENTIFICATION DIVISION statement.

To avoid confusion always place comments:

- After the IDENTIFICATION DIVISION statement that initiates the program to which they refer

and

- Before the END PROGRAM statement that terminates the program to which they refer.

Nesting: what the application programmer must do

1. Submit a top-level containing program and all its directly and indirectly contained programs as a single unit of compilation.
2. In each nested program that contains EXEC CICS commands, CICS built-in functions, or references to the EIB or COMMAREA, code DFHEIBLK and DFHCOMMAREA as the first two parameters of the PROCEDURE DIVISION header as follows:

```
PROCEDURE DIVISION USING DFHEIBLK
                        DFHCOMMAREA PARM1 PARM2 ...
```
3. In every call to a nested program that contains EXEC CICS commands, CICS built-in functions, or references to the EIB or COMMAREA, code DFHEIBLK and DFHCOMMAREA as the first two parameters of the CALL statement as follows:

```
CALL 'PROGA' USING DFHEIBLK
                        DFHCOMMAREA PARM1 PARM2 ...
```
4. For every call that forms part of the control hierarchy between the top-level program and a nested program that contains EXEC CICS commands, CICS built-in functions, or references to the EIB or COMMAREA, code DFHEIBLK and DFHCOMMAREA as the first two parameters of the CALL statement. In the PROCEDURE DIVISION in the called programs code DFHEIBLK and DFHCOMMAREA. This is necessary to allow addressability to the EIB and COMMAREA to be passed to programs not directly contained by the top-level program.
5. If it is not necessary to insert DFHEIBLK and DFHCOMMAREA in the PROCEDURE DIVISION of a nested program for any of the reasons given above (2, 3, and 4), calls to that program should *not* include DFHEIBLK and COMMAREA in the parameter list of the CALL statement.

An example of a nested program

A unit of compilation (see Figure 6 on page 40) consists of a top-level program W and three nested programs, X, Y, and Z, all directly contained by W.

Program W

During initialization and termination, calls Y and Z to do initial CICS processing and non-CICS file access. Calls X to do main processing.

Program X

Calls Z for non-CICS file access and Y for CICS processing.

Program Y

Issues CICS commands. Calls Z for non-CICS file access.

Program Z

Accesses files in batch mode.

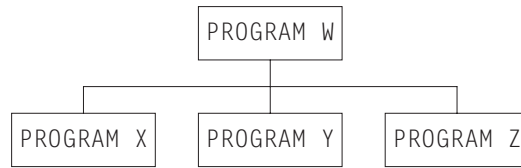


Figure 6. Nested program example—nesting structure

Applying the rules:

- Y must be **COMMON** to enable a call from X.
- Z must be **COMMON** to enable calls from X and Y.
- Y issues CICS commands, therefore:
 - All calls to Y must have **DFHEIBLK** and a **COMMAREA** as the first two parameters.
 - Y's **PROCEDURE DIVISION** header must have **DFHEIBLK** and **DFHCOMMAREA** as the first two parameters.
- Though X does not access the EIB or the communication area, it calls Y, which issues CICS commands. Therefore the call to X must have **DFHEIBLK** and a **COMMAREA** as the first two parameters and X's **PROCEDURE DIVISION** header must have **DFHEIBLK** and **DFHCOMMAREA** as its first two parameters.

Figure 7 on page 41 illustrates the points in “Nesting: what the application programmer must do” on page 39.


```

IDENTIFICATION DIVISION.
PROGRAM-ID. W.
.
.
PROCEDURE DIVISION.
.
.
    CALL Z.
.
.
    CALL Y USING DFHEIBLK COMMAREA.
.
.
    CALL X USING DFHEIBLK COMMAREA.
.
.
    IDENTIFICATION DIVISION.
    PROGRAM-ID. X.
.
.
    PROCEDURE DIVISION USING DFHEIBLK DFHCOMMAREA
.
.
    CALL Z.
.
.
    CALL Y USING DFHEIBLK COMMAREA.
.
.
    END PROGRAM X.
    IDENTIFICATION DIVISION.
    PROGRAM-ID. Y IS COMMON.
.
.
    PROCEDURE DIVISION USING DFHEIBLK DFHCOMMAREA.
.
.
    CALL Z.
.
.
    EXEC CICS...
.
.
    END PROGRAM Y.
    IDENTIFICATION DIVISION.
    PROGRAM-ID. Z IS COMMON.
.
.
    PROCEDURE DIVISION.
.
.
    END PROGRAM Z.
END PROGRAM W.

```

Figure 7. Nested program example: coding

Reference modification

Reference modification supports a method of referencing a substring of a character data item by specifying the starting (leftmost) position of the substring in the data item and, optionally, the length of the substring. The acceptable formats are:

```

data-name (leftmost-character-position:)
data-name (leftmost-character-position: length)

```

Data-name can be subscripted or qualified or both. Both *leftmost-character-position* and *length* can be arithmetic expressions. For more information about reference modification, qualification and subscripting, see the *Enterprise COBOL for z/OS Language Reference*, SC27-1408.

The translator accepts reference modification wherever the name of a character variable is permitted in a COBOL program or in an EXEC CICS command.

Note: If a CICS command uses reference modification in defining a data value, it should include a LENGTH option to specify the data length, unless the NOLENGTH translator option is used. Otherwise the translator generates a COBOL call with a LENGTH register reference in the form:

LENGTH OF (reference modification)

This is rejected by the compiler.

Global variables

The GLOBAL variable storage class is supported. A variable defined with the GLOBAL variable storage class in a top-level program (see “Translator action” on page 38) can be referred to in any of its nested programs, whether directly or indirectly contained.

The translator accepts the GLOBAL keyword.

Comma and semicolon as delimiters

A separator comma is a comma followed by a space. A separator semicolon is a semicolon followed by a space. A separator comma or a separator semicolon can be used as a separator wherever a space alone can be used.

The translator accepts the use in COBOL statements of a separator comma or a separator semicolon wherever a space can be used. For example, the translator accepts the statement:

IDENTIFICATION; DIVISION

The translator does **not** support the use of the separator comma and separator semicolon as delimiters in EXEC CICS commands. The only acceptable word delimiter in an EXEC CICS command continues to be a space.

Symbolic character definition

Symbolic characters can be defined in the SPECIAL-NAMES paragraph after the ALPHABET clause. A symbolic character is a program-defined word that represents a 1-character figurative constant.

The translator accepts the use of symbolic characters as specified in the standard.

Note: In general, the compiler does not accept the use of figurative constants and symbolic characters as arguments in CALL statements. For this reason, do not use figurative constants or symbolic constants in EXEC CICS commands, which are converted into CALL statements by the translator. There is one exception to this restriction: a figurative constant is acceptable in an EXEC CICS command as an argument to **pass** a value if it is of the correct data type. For example, a numeric figurative constant can be used in the LENGTH option.

Chapter 5. Programming in C and C++

CICS supports C and C++ programs that have been compiled by the supported compilers listed in the *CICS Release Guide*.

This chapter describes:

- “C and C++ programming considerations”
- “XPLink considerations for C and C++ programming” on page 46
- “Passing arguments in C or C++” on page 49
- “Accessing the EIB” on page 50
- “Locale support for C and C++” on page 51
- “Programming in C++” on page 51

C and C++ programming considerations

All the EXEC CICS commands available in COBOL, PL/I, and assembler language applications are also supported in C and C++ applications, with the exception of those commands related to nonstructured exception handling, see “Nonstructured exception handling” for more information.

C++ applications can also use the CICS C++ OO classes to access CICS services, instead of the EXEC CICS interface. See the *CICS C++ OO Class Libraries* manual, for more information about this interface.

Also note the following programming considerations:

Exception handling

nonstructured exception handling

The EXEC CICS commands related to nonstructured exception handling:

- HANDLE ABEND LABEL(label)
- HANDLE AID
- HANDLE CONDITION
- IGNORE CONDITION
- PUSH HANDLE
- POP HANDLE

are not supported in C and C++ applications. Use of these commands is diagnosed by the translator.

Condition handling

In a C or C++ application, every EXEC CICS command is treated as if it had the NOHANDLE or RESP option specified. This means that the set of “system action” transaction abends that result from a condition occurring but not being handled, is not possible in a C or C++ application. Control always flows to the next instruction, and it is up to the application to test for a normal response.

ABEND handling

HANDLE ABEND PROGRAM commands are allowed, but you cannot use PUSH HANDLE or POP HANDLE.

COMMAREA

The address of the communication area is not passed as an argument to a C or C++ main function. This means that C and C++ functions must use ADDRESS COMMAREA to obtain the address of the communications area.

EIB

The address of the EIB is not passed as an argument to a C or C++ main function. This means that C and C++ functions must use ADDRESS EIB to obtain the address of the EIB.

OVERFLOW conditions

If you want any OVERFLOW condition to be indicated in the RESP field on return from a SEND MAP command with the ACCUM option, you should specify the NOFLUSH option.

AMODE

All C and C++ language programs running under CICS **must** be link-edited with the attributes, AMODE(31), RMODE(ANY). They may reside above the 16MB line.

Invoking CSP programs

Consequently, when passing parameters to a program produced by the Cross-System Product (CSP) interactive application generator, you must either:

- Pass parameters below 16MB, or
- Re-link the CSP load library with AMODE(31).

Working storage

In C and C++, working storage consists of the stack and the heap. The location of the stack and heap, with respect to the 16MB line, is controlled by the ANYWHERE and BELOW options on the stack and heap run time options. The default is that both the stack and heap are located above the 16MB line.

Return value

If you terminate a C or C++ program with an **exit()** function or the **return** statement, instead of a CICS RETURN command, the value passed through the exit() function is saved in the EIBRESP2 field of the EIB on return from the program.

Note: If a program uses DPL to link to a program in another CICS region, EIBRESP2 values from the remote region are **not** returned to the program doing the DPL.

Sample programs

A set of sample application programs is provided in Table 1 to show how EXEC CICS commands can be used in a program written in the C or C++ language.

Table 1. Sample programs

Sample program	Map set	Map source	Transaction ID
DFH\$DMNU Operator instruction (3270)	DFH\$DGA	DFH\$DMA	DMNU
DFH\$DALL Update (3270)	DFH\$DGB	DFH\$DMB	DINQ, DADD, DUPD
DFH\$DBRW Browse (3270)	DFH\$DGC	DFH\$DMC	DBRW
DFH\$DREN Order entry (3270)	DFH\$DGK	DFH\$DMK	DORD
DFH\$DCOM Order entry queue print (3270)	DFH\$DGL	DFH\$DML	DORQ
DFH\$DREP Report (3270)	DFH\$DGD	DFH\$DMD	DREP

The transaction and program definitions are provided in group DFH\$DFLA in the CSD and should be installed using the command:

CEDA INSTALL GROUP(DFH\$DFLA)

The following record description files are provided as C or C++ language header files:

DFH\$DFIL—FILEA record descriptor
DFH\$DL86—L860 record descriptor

Data declarations

The following data declarations are provided by CICS for C and C++:

- Execution interface block definitions (EIB)
- BMS screen attributes definitions: C and C++ versions of the DFHBMSCA, DFHMSRCA, and DFHAID files are supplied by CICS, and may be included by the application programmer when using BMS.
- DLI support: a C language version of DFHDIB is included by the DLI translator if the translator option has been specified. (You have to include DLIUIB if the CALL DLI interface is used.)

The EIB declarations are enclosed in `#ifndef` and `#endif` lines, and are included in all translated files. The C or C++ compiler ignores duplicated declarations. The inserted code contains definitions of all the fields in the EIB, coded in C and C++.

Fetch function

Language Environment-conforming programs support the `fetch()` and `release()` functions. Modules to be fetched must be defined as PROGRAM resources to CICS, either explicitly or implicitly through `autoinstall`.

System function

CICS does not support the `system()` function, but two CICS commands, `LINK` and `XCTL`, provide equivalent function.

Macros

C and C++ do not support the use of CICS commands in macros.

Clock function

The `clock()` function returns a value (`time_t`) of -1.

Locale functions

All locale functions are supported for locales that have been defined in the CSD. CSD definitions for the IBM-supplied locales are provided in member `CEECCSD` of the `SCEESAMP` library. The `setlocale()` function returns `NULL` if the locale is not defined.

Debugging functions

The dump functions `csnap()`, `cdump()`, and `ctrace()` are supported. The output is sent to the CESE transient data queue. The dump cannot be written if the queue does not have a sufficient record length (`LRECL`). An `LRECL` of at least 161 is recommended.

iscics function

If you are adapting an existing program or writing a new program that is designed to run outside CICS as well as under CICS, the `iscics()` function may prove useful. It returns a non-zero value if your program is currently running under CICS, or zero otherwise. This function is an extension to the C library.

Restrictions

The following lists describe some of the restrictions that exist with C or C++ programs using Language Environment under CICS. You should check the relevant language guide for more specific details about those that apply to your installation:

- CICS does not support extended precision floating point.
- C and C++ languages do not support packed decimal data. The application has access to packed decimal data using the character string data type. No C or C++ standard library functions are available to perform arithmetic on this data, but you may write your own.
- You can easily use HOURS, MINUTES, and SECONDS options. You may define expiration times using TIME or INTERVAL options if you provide functions to handle them in your application.
- You can enter all CICS keywords in mixed case, except for CICS keywords on #pragma directives, which must be in upper case only.
- If you do not specify the LENGTH option on commands that support LENGTH (for example, READ, READNEXT, READPREV, and WRITE commands), the translator does not supply a default value. In effect, NOLENGTH is implicit for C programs.
- All native C and C++ functions are allowed in the source program, but the following functions are not recommended. Some are not executable and result in return codes or pointers indicating that the function has failed. Some may work but impact the performance or execution of CICS.
 - CDUMP
 - CSNAP
 - CTEST
 - CTRACE
 - CLOCK
 - CTDLI
 - SVC99
 - SYSTEM
 - SETLOCALE

For further information see the relevant *User's Guide*. Native C or C++ functions are implemented in the C or C++ runtime library.

- Native C or C++ file operations operate only on files opened with *type=memory* specified. I/O to CICS-supported access methods must use the CICS API.
- The string handling functions in the C or C++ standard library use a null character as an end-of-string marker. Because CICS does not recognize a null as an end-of-string marker, you must take care when using C or C++ functions, for example strcmp, to operate on CICS data areas.
- Two arguments, argc and argv, are normally passed to a C or C++ main function. argc denotes how many variables have been passed; argv is an array of zero-terminated variable strings. In CICS, the value of argc is 1, argv[0] is the transaction ID, and argv[1] is NULL.
- Where CICS expects a fixed-length character string such as a program name, map name, or queue name, you must pad the literal with blanks up to the required length if it is shorter than expected.

For EXEC DLI commands, the SEGMENT name is padded by the translator if a literal is passed.

- Take care not to use field names, which, though acceptable to the assembler, cause the C or C++ compiler toabend. These include \$, #, and @.

| XPLink considerations for C and C++ programming

| Extra Performance Linkage, (from here on it is abbreviated to XPLink), is a z/OS
 | feature which provides high performance subroutine call and return mechanisms.
 | This results in short and highly optimized execution path lengths.

Object Oriented programming is built upon the concept of sending 'messages' to objects which result in that object performing some actions. The message sending activity is implemented as a subroutine invocation. Subroutines, known as member functions in C++ terminology, are normally small pieces of code. The characteristic execution flow of a typical C++ program is of many subroutine invocations to small pieces of code. Programs of this nature benefit from the XPLink optimization technology.

MVS has a standard subroutine calling convention which can be traced back to the early days of System/360. This convention was optimized for an environment in which subroutines were more complex, there were relatively few of them, and they were invoked relatively infrequently. Object oriented programming conventions have changed this. Subroutines have become simpler but they are numerous, and the frequency of subroutine invocations has increased by orders of magnitude. This change in the size, numbers, and usage pattern, of subroutines made it desirable that the system overhead involved be optimized. XPLink is the result of this optimization.

PLEASE NOTE!

For z/OS 1.4 and above, and CICS TS 3.1 and above, the advice here that you **CAN** use the XPLINK compiler option with CICS application programs, overrides advice in z/OS and Language Environment manuals to the contrary.

z/OS and Language Environment manuals for C and C++ advise you that the XPLINK compiler option is not available to CICS application programs, because that used to be the case. Although these manuals are now being changed, you may be working with a copy of one of these manuals produced before this change.

The following topics cover:

- “XPLink, and the X8 and X9 TCBs”
- “Writing C and C++ programs, which are to be compiled with the XPLINK option, for the CICS environment” on page 48
- “Passing control between XPLink and non-XPLink objects” on page 48
- “Changing CICS definitions to obtain CICS support for objects compiled with the XPLINK option” on page 48
- “Global User exits and XPLink” on page 48

XPLink, and the X8 and X9 TCBs

CICS provides support for C and C++ programs compiled with the XPLINK option by using the multiple TCB feature in the CICS Open Transaction Environment (OTE) technology. X8 and X9 mode TCBs are defined to support XPLink tasks in CICS key and USER key. Each instance of an XPLink program uses one X8 or X9 TCB.

To use XPLink, your C or C++ application code must be re-entrant and threadsafe. The same code instance can be executing on more than one MVS TCB and, without threadsafe mechanisms to protect shared resources, the execution behavior of application code is unpredictable. This cannot be too strongly emphasized.

Writing C and C++ programs, which are to be compiled with the XPLINK option, for the CICS environment

The application developer is expected to do the following to take advantage of CICS XPLink support;

- Develop the code, strictly adhering to threadsafe programming principles and techniques
- Compile the C or C++ program with the XPLINK option set on
- Indicate in the PROGRAM resource definition that the program is threadsafe
- Consider the use of CICSVAR in CEEUOPT or in #pragma, (see note 3 in “Defining runtime options for Language Environment” on page 16 for details).

All programs using CICS XPLink support must be re-entrant and threadsafe. Only the application developer can guarantee that the code for a particular application satisfies these requirements.

Passing control between XPLink and non-XPLink objects

Each transfer of control from XPLink objects to non-XPLink objects, or the reverse, causes a switch between the QR TCB and an open TCB, (either an X8 or an X9 TCB). In performance terms, TCB switching is costly, you must take this performance overhead into account.

An XPLink object can invoke a non-XPLink object using either the EXEC CICS interface or the Language Environment interface.

A non-XPLink object can only invoke an XPLink object using the EXEC CICS interface. Use of the Language Environment interface for such invocations is not supported.

Changing CICS definitions to obtain CICS support for objects compiled with the XPLINK option

CICS support for programs compiled with the XPLINK option requires only that you show in the PROGRAM resource definition that the program is threadsafe. This indication, and the XPLink “signature” in the load module, are the only things required to put the task on an X8 or X9 TCB.

In the selection of a suitable TCB for a particular program, XPLink takes precedence over the existence of the OPENAPI value for the API attribute on the PROGRAM resource definition.

.

Global User exits and XPLink

XPCFTCH

When the exit XPCFTCH is invoked for a C or C++ program that was compiled with the XPLINK option, a flag is set indicating that any modified entry point address, if specified by the exit, will be ignored.

XPCTA

When the exit XPCTA is invoked for a C or C++ program that was compiled with the XPLINK option, a flag is set indicating that a resume address, if specified by the exit, will be ignored.

Other Global User exits are unaffected by XPLink support.

Passing arguments in C or C++

Arguments in C and C++ language are copied to the program stack at run time, where they are read by the function. These arguments can either be values in their own right, or they can be pointers to areas of memory that contain the data being passed. Passing a pointer is also known as passing a value *by reference*.

Other languages, such as COBOL and PL/I, usually pass their arguments *by reference*, which means that the compiler passes a list of addresses pointing to the arguments to be passed. This is the call interface supported by CICS. To pass an argument by reference, you prefix the variable name with **&**, unless it is already a pointer, as in the case when an array is being passed.

As part of the build process, the compiler may convert arguments from one data type to another. For example, an argument of type **char** may be converted to type **short** or type **long**.

When you send values from a C or C++ program to CICS, the translator takes the necessary action to generate code that results in an argument list of the correct format being passed to CICS. The translator does not always have enough information to enable it to do this, but in general, if the argument is a single-character or halfword variable, the translator makes a precall assignment to a variable of the correct data type and passes the address of this temporary variable in the call.

When you receive data from CICS, the translator prefixes the receiving variable name with **&**, which causes the C or C++ compiler to pass it values *by reference* rather than *by value* (with the exception of a character string name, which is left unchanged). Without the addition of **&**, the compiler would copy the receiving variable and then pass the address of the copy to CICS. Any promotion occurring during this copying could result in data returned by CICS being lost.

Table 2 shows the rules that apply when passing values as arguments in EXEC CICS commands.

Table 2. Rules for passing values as arguments in EXEC CICS commands

Data type	Usage	Coding the argument
Character literal	Data-value (Sender)	The user must specify the character literal directly. The translator takes care of any required indirection.
Character variable (char)	Data-area (Receiver)	The user must specify a pointer to the variable, possibly by prefixing the variable name with & .
Character variable (char)	Data-value (Sender)	The user must specify the character variable directly. The translator takes care of any required indirection.
Character string literal	Name (Sender)	The user can either code the string directly as a literal string or use a pointer which points to the first character of the string.
Character string variable	Data-area (Receiver) Name (Sender)	Whether receiving or sending, the argument should be the name of the character array containing the string—the address of the first element of the array.

Table 2. Rules for passing values as arguments in EXEC CICS commands (continued)

Data type	Usage	Coding the argument
Integer variable (short, long, or int)	Data-area (Receiver)	The user must specify a pointer to the variable, possibly by prefixing the variable name with & .
Integer variable (short, long, or int)	Data-value (Sender)	The user must specify the name of the variable. The translator looks after any indirection that is required.
Integer constant (short, long, or int)	Data-value (Sender)	The user must specify the integer constant directly. The translator takes care of any required indirection.
Structure or union	Data-area (Sender) Data-area (Receiver)	The user must code the address of the start of the structure or union, possibly by prefixing its name with & .
Array (of anything)	Data-area (Receiver) Data-value (Sender)	The translator does nothing. You must code the address of the first member of the array. This is normally done simply by coding the name of the array, which the compiler interprets as the address of the first member.
Pointer (to anything)	Ptr-ref (Receiver) Data-area (Sender)	Whether receiving or sending, the argument should be the name of the variable that denotes the address of interest. The translator takes care of the extra level of indirection that is necessary to allow CICS to update the pointer.
Note: Receiver is where data is being received from CICS; Sender is where data is being passed to CICS.		

Accessing the EIB

The address of the exec interface block (EIB) is not passed as an argument to a C or C++ main function. This means that C and C++ functions must use the ADDRESS EIB command to obtain the address of the EIB.

Addressability is achieved by using the command:

```
EXEC CICS ADDRESS EIB(dfheiptr);
```

or by passing the EIB address or particular fields therein as arguments to the CALL statement that invokes the external procedure.

If access to the EIB is required, an ADDRESS EIB command is required at the beginning of each program.

Naming EIB fields

Within a C or C++ application program, fields in the EIB are referred to in lower case and fully qualified as, for example, "dfheiptr->eibtrnid".

Data types in EIB fields

The following mapping of data types is used:

- Halfword binary integers are defined as "short int"
- Fullword binary integers are defined as "long int"
- Single-character fields are defined as "unsigned char"

- Character strings are defined as “unsigned char” arrays

Locale support for C and C++

```
# The CICS translator, by default, assumes that programs written in the C or C++
# language have been edited with the EBCDIC Latin-1 code page IBM-1047.

# If you have used an alternative code page, you can specify this in a pragma filetag
# directive at the start of the application program. The pragma statement must be the
# first non-comment statement in the program, and the filetag directive must be
# specified before any other directive in the pragma statement. The CICS translator
# scans for the presence of the filetag directive. The CICS translator only supports
# the default code page IBM-1047, the Danish EBCDIC code page IBM-277, the
# German EBCDIC code page IBM-273, and the Chinese EBCDIC code pages
# IBM-935 and IBM-1388.

# For example, if the program has been prepared with an editor using the German
# EBCDIC code page, it should begin with the following directive:
#
#      ??=pragma filetag ("IBM-273")

# If your application program uses a mix of different code pages (for example, if you
# are including header files edited in a code page different to that used for the
# ordinary source files), all of the files must include the pragma filetag directive, even
# if they are in the default code page IBM-1047.

# Some older IBM C compilers which are no longer in service, but can still be used
# with the CICS translator, might not support the use of the pragma filetag directive.
# Check the documentation for your compiler if you are not sure whether your
# compiler supports this. All the IBM C/C++ compilers that are listed in the topic
# “High-level language support” in the CICS Transaction Server for z/OS Release
# Guide which are still in service support the use of the pragma filetag directive.
```

Programming in C++

C++ supports object-oriented programming and you can use this language in the same way as you would use the C language. You must specify that the translator is to translate C++ using the CPP option.

C++ programs must also be defined with the LANGUAGE(LE370) option. See Chapter 3, “Language Environment,” on page 9 for information about this environment.

Restrictions

C++ uses “//” for single line comments. Do not put a comment in the middle of an EXEC CICS command. For instance, this example does not work:

```
EXEC CICS SEND TEXT FROM(errmsg)
      LENGTH(msglen) // Send error message to screen
      RESP(rcode)
      RESP2(rcode2);
```

These examples are valid:

```
EXEC CICS SEND TEXT FROM(errmsg)
      LENGTH(msglen)
      RESP(rcode)
      RESP2(rcode2);      //Send error message to screen
```

```
EXEC CICS SEND TEXT FROM(errmsg)
      LENGTH(msglen)      /* Send error message to screen */
      RESP(rcode)
      RESP2(rcode2);
```

Chapter 6. Programming in PL/I

CICS supports PL/I programs that have been compiled by the supported compilers listed in the *CICS Release Guide*. All PL/I programs are executed under the runtime support provided by Language Environment.

Runtime options, if needed, can be specified in a **plixopt** character string. See “Defining runtime options for Language Environment” on page 16 and the *Language Environment Customization* manual or information about customizing runtime options.

If you are converting a program that was previously compiled with a non-Language Environment conforming compiler, you must ensure that neither NOSTAE nor NOSPIE is specified in a **plixopt** string, because this will cause Language Environment to set TRAP (OFF). TRAP (ON) must be in effect for applications to run successfully.

If OPTIONS(MAIN) is specified in an application program, that program can be the first program of a transaction, or control can be passed to it by means of a LINK or XCTL command.

In application programs where OPTIONS(MAIN) is not specified, it cannot be the first program in a transaction, nor can it have control passed to it by an LINK or XCTL command, but it can be link-edited to a main program.

This chapter describes:

- “PL/I programming restrictions”
- “Language Environment considerations for PL/I applications” on page 54

PL/I programming restrictions

The following restrictions apply to a PL/I program that is to be used as a CICS application program.

- You cannot use the multitasking built-in functions:

COMPLETION
PRIORITY
STATUS

- You cannot use the multitasking options:

EVENT
PRIORITY
TASK

- You should not use the PL/I statements:

CLOSE	DELAY
DELETE	DISPLAY
EXIT	GET
HALT	LOCATE
OPEN	PUT
READ	REWRITE
STOP	WRITE
UNLOCK	

The following commands are supported :

FETCH
RELEASE

You are provided with EXEC CICS commands for the storage and retrieval of data, and for communication with terminals. (However, you can use CLOSE, PUT, and OPEN, for SYSPRINT.)

- You cannot use PL/I Sort/Merge.
- You cannot use static storage (except for read-only data).
- If you declare a variable with the STATIC attribute and EXTERNAL attribute you should also include the INITIAL attribute. If you do not, such a declaration generates a common CSECT that cannot be handled by CICS.
- You cannot use the PL/I 48-character set option in EXEC CICS statements.
- Do not define variables or structures with variable names that are the same as variable names generated by the translator. These begin with DFH. Care must be taken with the LIKE keyword to avoid implicitly generating such variable names.
- All PROCEDURE statements must be in upper case, with the exception of the PROCEDURE name, which may be in lower case.
- The suboptions of the XOPTS option of the *PROCESS statement must be in upper case.
- If a CICS command uses the SUBSTR built-in function in defining a data value, it should include a LENGTH option to specify the data length, unless the translator option NOLENGTH is specified. If it does not, the translator generates a PL/I call including an invocation of the CSTG built-in function in the form:

CSTG(SUBSTR(...))

This is rejected by the compiler.

- For Enterprise PL/I Version 3 Release 2 or later, specify either the FLOAT(NOAFP) compiler option, or the FLOAT(AFP(VOLATILE)) compiler option.
 - If your program makes use of floating-point sparingly, specify the FLOAT(NOAFP) option. The application will only use the traditional 4 floating point registers, and has less work to do when saving registers.
 - If your program makes significant use of floating-point, specify the FLOAT(AFP(VOLATILE)) option. The application will use all 16 floating point registers, but will have more work to do when saving registers.

Language Environment considerations for PL/I applications

The advice and restrictions listed in Chapter 6, “Programming in PL/I,” on page 53 apply to PL/I programs running under Language Environment.

Language Environment-conforming PL/I programs can CALL a program that appears in a FETCH or RELEASE statement and can RELEASE it subsequently.

There are some restrictions on the PL/I for MVS & VM statements that can be used in a fetched procedure. These restrictions are described in *PL/I MVS & VM Language Reference*. Many of the restrictions have been removed with VisualAge PL/I. See the *VisualAge PL/I for OS/390 Compiler and Run-Time Migration Guide*.

To enable a PL/I procedure to be fetched, code the option FETCHABLE in the OPTIONS on the PROCEDURE statement. This option indicates that the procedure can **only** be invoked dynamically. An OPTIONS(MAIN) procedure cannot be fetched; FETCHABLE and MAIN are mutually exclusive options. Treat the FETCHABLE procedure like a normal CICS program: that is, link-edited with any required subroutines, placed in the CICS application program library, defined, and installed as a program, either in the CSD or using program autoinstall.

No special considerations apply to the use of FETCH when both the fetching and the fetched programs have the same AMODE attribute. Language Environment, however, also supports the fetching of a load module that has an AMODE attribute different to the program issuing the FETCH. In this case, Language Environment performs the AMODE switch, and the following constraints apply:

- If any fetched module is to execute in 24-bit addressing mode, the fetching module must have the RMODE(24) attribute regardless of its AMODE attribute.
- Any variables passed to a fetched routine must be addressable in the AMODE of the fetched procedure.

Communicating between modules compiled with different compilers

You can link-edit non-Language Environment-conforming PL/I subroutines with a Language Environment-conforming main program. Static calls are supported from any version of PL/I, but dynamic calls are supported only from Language Environment-conforming procedures. Called subroutines can issue CICS commands if the address of the EIB is available in the subroutine. You can achieve this either by passing the address of the EIB to the subroutine, or by coding EXEC CICS ADDRESS EIB(DFHEIPTR) in the subroutine before issuing any other CICS commands.

Entry point

CEESTART is the only entry point for PL/I applications running under Language Environment. This entry point is set for programs compiled using Language Environment-conforming compilers. You can re-link object modules produced by non-Language Environment-conforming compilers for running under Language Environment by using the following linkage-editor statements:

```
INCLUDE SYSLIB(CEESTART)
INCLUDE SYSLIB(CEESG010)
INCLUDE SYSLIB(DFHELII)
REPLACE PLISTART
CHANGE PLIMAIN(CEEMAIN)
INCLUDE mainprog
INCLUDE subprog1
.....
.....
ORDER CEESTART
ENTRY CEESTART
NAME progname(R)
```

The INCLUDE statement for the object modules must come immediately after the CHANGE statement and there is also a requirement under Language Environment that the main program must be included before any subroutines. (This requirement did not exist for modules produced by non-conforming compilers.) For Enterprise PL/I programs that are compiled with OPTIONS(FETCHABLE), the binder ENTRY statement must be the name of the PROCEDURE.

Re-link utility for PL/I

If you have only the load module for a CICS program compiled by a non-conforming compiler, there is a file of linkage editor input, IBMWRLKC, specifically for CICS programs, located in the sample library SCEESAMP, to replace OS PL/I library routines in a non-conforming executable program with Language Environment routines. For more information about using IBMWRLKC, see the *PL/I MVS & VM V1R1.1 Compiler & Runtime Migration Guide*.

Abend codes

If a CICS PL/I program abends under Language Environment, your CICS abend handlers are given a Language Environment abend code, rather than a PL/I abend code. To avoid changing your programs, you can modify the sample

user condition handler, CEEWUCHA, supplied by Language Environment in the SCEESAMP library. This user condition handler can be made to return PL/Iabend codes instead of the Language Environment codes. Use the USRHDLR runtime option to register it to do this. For details of this option see the *z/OS Language Environment Programming Guide*.

Chapter 7. Programming in Assembler

CICS supports assembler programs that have been compiled by the supported compilers listed in the *CICS Release Guide*. All assembler programs are executed under the runtime support provided by Language Environment.

Note the following Assembler language programming considerations:

Language restrictions

1. The following instructions cannot be used in an assembler language program that is to be used as a CICS application program:

COM	Identify blank common control section.
ICTL	Input format control.
OPSYN	Equate operation code.

2. The following additional restrictions apply if an assembler language program is to be translated with the LEASM option, see “LEASM” on page 78.

- Register 2 cannot be used as a code base register.
- Register 12 is reserved by Language Environment to point to the Language Environment common anchor area (CAA) and so cannot be used at all by the program without being saved and restored as appropriate.
- Register 13 must be used as the one and only working storage base register.
- The program cannot be a Global User Exit program (GLUE) or a Task-Related User Exit program (TRUE).
- The program must not use, or depend on, any AMODE(24) code.

BAKR

When using BAKR instructions (branch and stack) to provide linkage between assembler programs, take care that the linked-to program does not issue EXEC CICS requests. If CICS receives control and performs a task switch before the linked-to program returns by a PR instruction (program return), then other tasks might be dispatched and issue further BAKR / PR calls. These modify the linkage-stack and result in the wrong environment being restored when the original task issues its PR instruction.

Working storage

Working storage is allocated either above or below the 16MB line, according to the value of the DATALOCATION parameter on the PROGRAM definition in the CSD.

HANDLE ABEND LABEL

CICS does not allow the use of HANDLE ABEND LABEL in Assembler programs that do not use DFHEIENT and DFHEIRET. Assembler programs that use the Language Environment stub CEESTART should either use HANDLE ABEND PROGRAM or a Language Environment service such as CEEHDLR.

31-bit addressing

The following restriction applies to an assembler language application program executing in 31-bit mode:

- The COMMAREA option is restricted in a mixed addressing mode transaction environment. For a discussion of the restriction, see “Using mixed addressing modes” on page 571.

MVS restrictions

The following restrictions apply to an assembler language application program that uses access registers to exploit the extended addressability of ESA/370 processors:

- You must be in primary addressing mode when invoking any CICS service. The primary address-space must be the home address-space. All parameters passed to CICS must reside in the primary address-space.
- CICS does not *always* preserve access registers. You must save them before you invoke a CICS service, and restore them before using the access registers again.

For more guidance information about using access registers, see the z/OS: MVS Programming: Extended Addressability Guide.

64-bit registers

The following restriction applies to an assembler language application program that uses 64-bit registers to exploit 64-bit addressing mode or 64-bit binary operations:

- CICS does not *always* preserve the high order words of 64-bit registers. You must save them before you invoke a CICS service, and restore them before using the 64-bit registers again.

For more guidance information about using 64-bit addressing mode and 64-bit binary operations, see the z/OS: *MVS Programming: Assembler Services Guide*.

Language Environment considerations for Assembler applications

Like HLL programs, assembler programs are classified as either conforming or non-conforming with respect to Language Environment. For assembler programs, conformance depends on the linkage and register conventions observed, rather than the assembler used. By definition, a Language Environment-conforming assembler routine is defined as one coded using the CEEENTRY and associated Language Environment macros.

Conformance governs the use of assembler programs by call from an HLL program. Both conforming and non-conforming assembler subroutines may be called either statically or dynamically from C or C++, COBOL or PL/I. However, there are differences in register conventions and other requirements for the two types. These are described below. Rules for mixing languages, including assembler, are discussed in “Mixing languages in Language Environment” on page 13.

Conforming MAIN programs

If you are coding a new assembler MAIN program that you want to conform to the Language Environment interface or if your assembler routine calls Language Environment services, observe the following:

- Use the macros provided by Language Environment. For a list of these macros, see the *z/OS Language Environment Programming Guide*.
- Ensure that the CEEENTRY macro contains the option MAIN=YES. (MAIN=YES is the default).

- Translate your assembler routine with *ASM XOPTS(LEASM) or, if it contains CICS commands, with *ASM XOPTS(LEASM NOPROLOG NOEPILOG).

Conforming sub-routines programs

If you are coding a new assembler sub-routine that you want to conform to the Language Environment interface or if your assembler routine calls Language Environment services, observe the following:

- Use the macros provided by Language Environment. For a list of these macros, see the *z/OS Language Environment Programming Guide*.
- Ensure that the CEEENTRY macro contains the option MAIN=NO. (MAIN=YES is the default).
- Translate your assembler routine with *ASM XOPTS(NOPROLOG NOEPILOG) if it contains CICS commands.
- Ensure that the CEEENTRY macro contains the option NAB=NO if your routine is invoked by a static call from VS COBOL II. (NAB is **N**ext **A**vailable **B**yte (of storage). NAB=NO means that this field may not be available, so the CEEENTRY macro generates code to find the available storage.)

All conforming routines

To communicate properly with assembler routines, observe certain register conventions on entry to the assembler routine, while it is running, and on exit from the assembler routine.

Entry

On entry into a Language Environment-conforming assembler subroutine, these registers must contain the following values when NAB=YES is specified on the CEEENTRY macro:

R0	Reserved
R1	Address of the parameter list, or zero
R12	Common anchor area (CAA) address
R13	Caller's dynamic storage area (DSA)
R14	Return address
R15	Entry point address

Language Environment-conforming HLLs generate code that follows these register conventions, and the supplied macros do the same when you use them to write your Language Environment-conforming assembler routine. On entry to an assembler routine, CEEENTRY saves the caller's registers (R14 through R12) in the DSA provided by the caller. It allocates a new DSA and sets the NAB field correctly in this new DSA. The first half word of the new DSA is set to binary zero and the back chain in the second word is set to point to the caller's DSA.

While the subroutine is running

R13 must point to the routine's DSA at all times while the Language Environment-conforming assembler routine is running.

At any point in your code where you CALL another program, R12 must contain the common anchor area (CAA) address, except in the following cases:

- When calling a COBOL program

- When calling an assembler routine that is not Language Environment-conforming
- When calling a Language Environment-conforming assembler routine that specifies NAB=NO on the CEEENTRY macro

Exit

On exit from a Language Environment-conforming assembler routine, R0, R1, R14, and R15 are undefined. All the other registers must have the contents they had upon entry.

The CEEENTRY macro automatically sets a module to AMODE (ANY) and RMODE (ANY). If you are converting an existing assembler routine to be Language Environment-conforming and the routine contains data management macros coded using 24-bit addressing mode, then you should change the macros to use 31-bit mode. If it is not possible to change all the modules making up a program to use 31-bit addressing mode, and if none of the modules explicitly sets RMODE (24), then you should set the program to be RMODE (24) during the link-edit process.

Non-conforming routines

Observe the following conventions when running non-Language Environment-conforming subroutines under Language Environment:

- R13 must contain the address of the executing routine's register save area.
- The first two bytes of the register save area must be binary zeros.
- The register save area back chain must be set to a valid 31-bit address (the high-order byte must be zero if it is a 24-bit address).

If your assembler routine relies on C/C++, COBOL, or PL/I control blocks (for example, a routine that tests flags or switches in these control blocks), check that these control blocks have not changed under Language Environment. For more information, see the appropriate Language Environment migration guide.

Non-conforming routines cannot use Language Environment callable services.

Note: CICS does not allow the use of HANDLE ABEND LABEL in Assembler programs that do not use DFHEIENT and DFHEIRET. Assembler programs that use the Language Environment stub CEESTART should either use HANDLE ABEND PROGRAM or a Language Environment service such as CEEHDLR. See "Using Language Environment Abend-handling" on page 12 for information about CEEHDLR.

For more information or for explanations of the terms used in this section see the chapter on "Assembler Considerations" in the *z/OS Language Environment Programming Guide*.

Calling Assembler programs

Assembler language application programs that contain commands can have their own RDO program definition. Such programs can be invoked by COBOL, C or C++, PL/I, or assembler language application programs using LINK or XCTL commands (see Chapter 45, "Program control," on page 565). However, because programs that contain commands are invoked by a system standard call, they can also be invoked by a COBOL, C, C++, or PL/I CALL statement or by an assembler language CALL macro.

A single CICS application program, as defined in an RDO program definition, may consist of separate CSECTs compiled or assembled separately, but linked together.

An assembler language application program that contains commands can be linked with other assembler language programs, or with programs written in one or more high-level languages (COBOL, C, C++, or PL/I). For details of mixing languages in an application load module, refer to *z/OS Language Environment Writing Interlanguage Communications Applications*.

If an assembler language program (that is link edited separately) contains command-level calls, and is called from a high-level language program, it requires its own CICS interface stub. If the assembler program is link edited with the high-level language program that calls it, then the assembler program does not need a stub, but if you provide one, the message MSGIEW024I is issued, but this can be ignored.

Because assembler language application programs containing commands are always passed the parameters EIB and COMMAREA when invoked, the CALL statement or macro must pass these two parameters followed, optionally, by other parameters.

For example, the PL/I program in file PLITEST PLI calls the assembler language program ASMPROG, which is in file ASMTEST ASSEMBLE. The PL/I program passes three parameters to the assembler language program, the EIB, the COMMAREA, and a message string.

```
PLIPROG:PROC OPTIONS(MAIN);
  DCL ASMPROG ENTRY EXTERNAL;
  DCL COMA CHAR(20), MSG CHAR(14) INIT('HELLO FROM PLI');
  CALL ASMPROG(DFHEIBLK,COMA,MSG);
  EXEC CICS RETURN;
END;
```

Figure 8. PLITEST PLI

The assembler language program performs an EXEC CICS SEND TEXT command, which displays the message string passed from the PL/I program.

```
DFHEISTG DSECT
MSG      DS    CL14
MYRESP   DS    F
ASMPROG  CSECT
  L       5,8(1)
  L       5,0(5)
  MVC     MSG,0(5)
  EXEC CICS SEND TEXT FROM(MSG) LENGTH(14) RESP(MYRESP)
END
```

Figure 9. ASMTEST ASSEMBLE

You can use JCL procedures supplied by CICS to compile and link the application, as follows:

1. Assemble and link ASMTEST using the DFHEITAL procedure:

```
//ASMPROG EXEC DFHEITAL
//TRN.SYSIN DD *
.... program source ...
/*
//LKED.SYSIN DD *
  NAME ASMTEST(R)
/*
```

2. Compile and link PLITEST using the DFHYITPL procedure, and provide linkage editor control statements that include the ASMTEST load module created by the DFHEITAL procedure:

```

//PLIPROG EXEC DFHYITPL
//TRN.SYSIN DD *
    .... program source ...
/*
//LKED.SYSIN DD *
    INCLUDE SYSLIB(ASMTEST)
    ENTRY CEESTART
    NAME PLITEST(R)
/*

```

Note: Step 2 assumes that the ASMTEST load module created by DFHEITAL was stored in a library included in the SYSLIB dataset concatenation.

The load module created by the DFHYITPL procedure includes both the DFHEAI stub (included by DFHEITAL) and the DFHELII stub (included by DFHYITPL). This causes the linkage editor or binder program to issue a warning message because both stubs contain an entry point named DFHEII. This message can be ignored.

The DFHEAI stub must be included at the beginning of the program in the output
from the link edit. To achieve this, ORDER and INCLUDE statements for DFHEAI
must be in the link-edit step of your JCL. When you use the CICS-supplied
assembler procedure DFHEITAL in the SDFHPROC library to translate, assemble,
and link-edit application programs written in assembler language, the COPYLINK
step of this procedure copies SDFHMAC(DFHEILIA). DFHEILIA contains the
following statements that must be included:

```

#         ORDER DFHEAI
#         INCLUDE SYSLIB(DFHEAI)

```

The statements are put into a temporary file that is concatenated before the
assembled application program in the LKED step of the procedure.

If you are writing your own JCL, you only need to include the DFHELII stub, because this contains the entry points needed for all languages.

An assembler language application program that is called by another begins with the DFHEIENT macro and ends with the DFHEIRET macro. The CICS translator inserts these for you, so if the program contains EXEC CICS commands and is to be passed to the translator, as in the example just given, you do not need to code these macros.

Part 2. Preparing applications to run

Chapter 8. Translation and compilation	67
The integrated CICS translator	67
Using the integrated CICS translator	68
Specifying CICS translator options	68
The translation process	69
The CICS-supplied translators	72
Dynamic invocation of the separate translator	72
Translator option list	72
Data definition (DD name) list	72
Using a CICS translator	73
Defining translator options	74
Translator options	75
APOST	75
CBLCARD	75
CICS	75
COBOL2	75
COBOL3	75
CPP	76
CPSM	76
DBCS	76
DEBUG	76
DLI	76
EDF	76
EPILOG	76
EXCI	76
FEPI	77
FLAG (I, W, E, or S)	77
GDS	77
GRAPHIC	77
LEASM	78
LENGTH	78
LINECOUNT(n)	78
LINKAGE	78
MARGINS(m,n[,c])	78
NATLANG(EN or KA)	79
NOCBLCARD	79
NOCPSM	79
NODEBUG	79
NOEDF	80
NOEPILOG	80
NOFEPI	80
NOLENGTH	80
NOLINKAGE	80
NONUM	80
NOOPSEQUENCE	80
NOOPTIONS	81
NOPROLOG	81
NOSEQ	81
NOSEQUENCE	81
NOSOURCE	81
NOSPIE	81
NOVBREF	81
NUM	81

OPMARGINS(m,n[,c])	82
OPSEQUENCE(m,n).	82
OPTIONS.	82
PROLOG	82
QUOTE	82
SEQ.	83
SEQUENCE(m,n)	83
SOURCE	83
SP	83
SPACE(1 or 2 or 3)	83
SPIE	83
SYSEIB	83
VBREF.	84
Translator options table.	85
Using COPY statements	86
The CICS-supplied interface modules	86
The EXEC interface modules.	86
The CPI Communications interface module	86
The SAA Resource Recovery interface module	86
Using the EXEC interface modules	87
COBOL	88
PL/I	88
C and C++	89
Assembler language	89
EXAMPLE Assembler language PROGRAM using LEASM.	89

Chapter 9. Installing application programs.	99
Program installation roadmap	99
Preparing for program installation	100
Defining MVS residence and addressing modes	100
Establishing a program's addressing mode	101
CICS address space considerations.	101
Making programs permanently resident	102
Running applications in the link pack area	102
Running application programs in the RDSAs	103
Assembler	103
C and C++.	104
COBOL	104
PL/I	105
Using BMS map sets in application programs	105
Using the CICS-supplied procedures to install application programs	106
Installing programs in load library secondary extents	108
Including the CICS-supplied interface modules.	108
Installing assembler language application programs	109
Installing COBOL application programs	110
Sample JCL to install COBOL application programs	111
Installing PL/I application programs	114
Sample JCL to install PL/I application programs	114
PL/I procedure with an integrated translator	115
Installing C application programs	117
Sample JCL to install C application programs	118
Including pre-translated code with your C source code	119
Using your own job streams	120
Translator requirements	120
Online programs that use EXEC CICS or EXEC DLI commands	120
Online programs that use the CALL DLI interface.	122

Batch or BMP programs that use EXEC DLI commands	123
Batch or BMP programs that use DL/I CALL commands	123
Chapter 10. Installing map sets and partition sets	125
Installing map sets	126
Types of map sets	126
Defining the type of map set you require	127
Using extended data stream terminals	127
Installing physical map sets	128
Installing symbolic description map sets	129
Installing physical and symbolic description maps together	131
Using the DFHMAPT procedure to install HTML templates from BMS maps	131
JCL to install physical and symbolic description maps	132
Adding a CSECT to your map assembly	132
Installing partition sets	133
Defining programs, map sets, and partition sets to CICS	134

Chapter 8. Translation and compilation

Most older compilers (and assemblers) cannot process CICS commands directly. This means that an additional step is needed to convert your program into executable code. This step is called **translation**, and consists of converting CICS commands into the language in which the rest of the program is coded, so that the compiler (or assembler) can understand them.

Modern compilers can use the **integrated CICS translator** approach, where the compiler interfaces with CICS at compile time to interpret CICS commands and convert them automatically to calls to CICS service routines. If you use the integrated CICS translator approach then many of the translation tasks described in “The translation process” on page 69 are done at compile time for you, and you do not need to execute the additional translator step.

This section describes:

- “The integrated CICS translator”
- “The translation process” on page 69
- “The CICS-supplied translators” on page 72
- “Using a CICS translator” on page 73
- “Defining translator options” on page 74
- “Using COPY statements” on page 86
- “The CICS-supplied interface modules” on page 86
- “Using the EXEC interface modules” on page 87

The integrated CICS translator

In earlier CICS releases, CICS application programs had to be translated before they could be compiled. The translators find EXEC CICS commands, make them into comments, and generate CALLs appropriate to the language. The CICS-supplied jobs for compiling user application programs all contain an initial job step that invokes the translator appropriate to the compiler invoked in the following job step.

The CICS-supplied separate translators change the line numbers in source programs, which means that an intermediate listing, with the translator-generated CALLs, which must be used when debugging an application program. With the integrated translator, application development is made easier because there is only one listing — the original source statements, and the CICS error messages are included in the compiler listing. The process of translating and compiling is also less error-prone because it is no longer necessary to translate included members separately.

The Language Environment-conforming language compilers that support the integrated CICS translator scan the application source and call the integrated CICS translator at relevant points.

The releases of the COBOL and PL/I compilers which support the CICS integrated translator are listed in the *CICS Release Guide*. The integrated translator is supported in z/OS V1.7 XL C/C++ and later compilers. If you use any other compiler, including Assembler, you will need to translate your program in the traditional way described in Chapter 8, “Translation and compilation.”

|

|
|

Using the integrated CICS translator

The language compilers provide various procedures that you can use with the integrated CICS translator. They are documented in the Programming Guides for Enterprise PL/I for z/OS and for XL C/C++.

The procedure that you use needs to have CICSTS31.CICS.SDFHLOAD added to the STEPLIB concatenation for the compile step and the link-edit step should include the interface module DFHELII at the start of the step.

To use the integrated CICS translator for PL/I you must specify the compiler option SYSTEM(CICS).

To use the integrated CICS translator for COBOL, the compiler options CICS, LIB, NODYNAM, and RENT must be in effect. NODYNAM is not a restriction specific to the integrated translator. DYNAM is not supported for code that is separately translated and compiled. Do not use SIZE(MAX), because storage must be left in the user region for integrated CICS translator services. Instead, use a value such as SIZE(4000K), which should work for most programs.

If you are running DB2 Version 7 or later and preparing a COBOL program using a compiler with integrated translator, the compiler also provides an SQL statement coprocessor (which produces a DBRM), so you do not need to use a separate DB2 precompiler. See the *CICS DB2 Guide* and the *DB2 for OS/390 and z/OS Application Programming and SQL Guide* for more information on using the SQL statement coprocessor.

To use the integrated CICS translator for C and C++, use the CICS option.

Specifying CICS translator options

To specify CICS translator options when using the XL C/C++ compiler specify the compiler option, CICS, with the translator options inside parentheses. For example:

CICS(opt1 opt2 optn ...)

To specify CICS translator options when using the PL/I compiler specify the compiler option, PP(CICS), with the translator options enclosed in apostrophes and inside parenthesis. For example:

PP(CICS('opt1 opt2 optn ...'))

For more information on specifying PL/I compiler options see the *Enterprise PL/I for z/OS and OS/390 Programming Guide*.

To specify CICS translator options when using the COBOL compiler specify the compiler option, CICS, with the translator options enclosed in apostrophes and inside parenthesis. For example:

CICS('opt1 opt2 optn ...')

Note: The XOPTS translator option must be changed to the CICS compiler option. XOPTS is not accepted when using the integrated CICS translator.

For more information on specifying COBOL compiler options see the *Enterprise COBOL for z/OS and OS/390: Programming Guide*.

For a description of all of the translator options see “Defining translator options” on page 74.

Many of the translator options, such as those associated with translator listings, do not apply when using the integrated CICS translator. These options, if specified, are ignored. The EXCI option is not supported, the CICS option is assumed.

The translator options that can be used effectively with the integrated CICS translator are:

- APOST or QUOTE
- CPSM or NOCPSM
- CICS
- DBCS
- DEBUG or NODEBUG
- DLI
- EDF or NOEDF
- FEPI or NOFEPI
- GRAPHIC
- LENGTH or NOLENGTH
- LINKAGE or NOLINKAGE
- NATLANG
- SP
- SYSEIB

The translation process

For compilers without integrated translators, CICS provides a translator program for each of the languages in which you may write, to handle both EXEC CICS and EXEC DLI statements.

For compilers with integrated translators, the compilers interface with CICS to handle both EXEC CICS and EXEC DLI statements.

A language translator reads your source program and creates a new one; most normal language statements remain unchanged, but CICS commands are translated into CALL statements of the form required by the language in which you are coding. The calls invoke CICS-provided "EXEC" interface modules, which later get link-edited into your load module, and these in turn invoke the requested services at execution time.

There are three steps: translation, compilation (assembly), and link-edit. Figure 10 on page 70 shows these 3 steps.

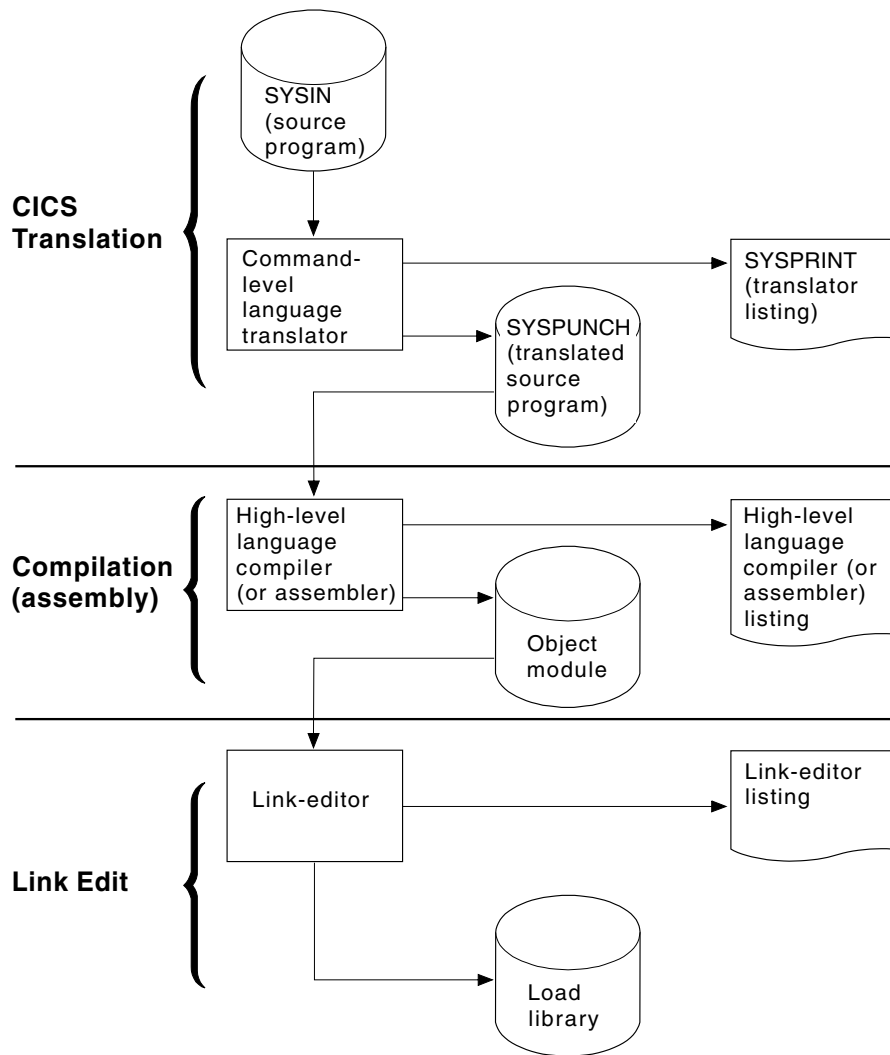


Figure 10. Preparing an application program

The translators for all languages use one input and two output files:

SYSIN (Translator input) is the file that contains your source program.

If the SYSIN file is defined as a fixed blocked data set, the maximum record length that the data set can possess is 80 bytes. Passing a fixed blocked data set with a record length of greater than 80 bytes to the translator results in termination of translator execution. If the SYSIN file is defined as a variable blocked data set, the maximum record length that the data set can possess is 100 bytes. Passing a variable blocked data set with a record length greater than 100 bytes to the translator causes the translator to stop with an error.

SYSPUNCH

(Translated source) is the translated version of your source code, which becomes the input to the compile (assemble) step. In this file, your source has been changed as follows:

- The EXEC interface block (EIB) structure has been inserted.
- EXEC CICS, EXEC CPSM and EXEC DLI commands have been turned into function call statements.

- CICS DFHRESP, EYUVALUE, and DFHVALUE built-in functions have been processed.
- A data interchange block (DIB) structure and initialization call have been inserted if the program contains EXEC DLI statements.

The CICS commands that get translated still appear in the source, but as comments only. Generally the non-CICS statements are unchanged. The output from the translator always goes to an 80 byte fixed-record length data set.

SYSPRINT

(Translator listing) shows the number of messages produced by the translator, and the highest severity code associated with any message. The options used in translating your program also appear, unless these have been suppressed with the NOOPTIONS option.

For COBOL, C, C++, and PL/I programs, SYSPRINT also contains the messages themselves. In addition, if you specify the SOURCE option of the translator, you also get an annotated listing of the source in SYSPRINT. This listing contains almost the same information as the subsequent compilation listing, and therefore many installations elect to omit it (the NOSOURCE option). One item you may need from this listing which is not present in the compile listing, however, is the line numbers, if the translator is assigning them. Line numbers are one way to indicate points in the code when you debug with the execution diagnostic facility (EDF). If you specify the VBREF option, you also get a list of the commands in your program, cross-referenced by line number, and you can use this as an alternative to the source listing for EDF purposes.

For assembler language programs, SYSPRINT contains only the translator options, the message count and maximum severity code. The messages themselves are inserted into the SYSPUNCH file as comments after the related statement. This causes the assembler to copy them through to the assembler listing, where you can check them. You may also see MNOTES that are generated by the assembler as the result of problems encountered by the translator.

Note: If you use EXEC SQL, you need additional steps to translate the SQL statements and bind; see the *Application Programming and SQL Guide* for information about these extra steps.

CICS provides a procedure to execute these steps in sequence for each of the languages it supports. “Using the CICS-supplied procedures to install application programs” on page 106 describes how to use these procedures, and exactly what they do.

You can control the translation process by specifying a number of **options**. For example, if your program uses EXEC DLI calls, you need to tell the translator.

The translator may produce error messages, and it is as important to check these messages as it is to check the messages produced by the compiler and link-editor. See “The CICS-supplied translators” on page 72 for the location of these messages.

EXEC commands are translated into CALL statements that invoke CICS interface modules. These modules get incorporated into your object module in the link-edit step, and you see them in your link-edit output listing. You can read more about these modules in “The CICS-supplied interface modules” on page 86.

The CICS-supplied translators

The following CICS-supplied translators are installed in the CICSTS31.CICS.SDFHLOAD library:

Assembler DFHEAP1\$
C DFHEDP1\$
COBOL DFHECP1\$
PL/I DFHEPP1\$

Dynamic invocation of the separate translator

You can invoke the command-level language translator dynamically from a batch assembler-language program using an ATTACH, CALL, LINK, or XCTL macro; or from a C, PL/I, or COBOL program using CALL. If you use ATTACH, LINK, or XCTL, use the appropriate translator load module, DFHE x P1\$, where $x=A$ for assembler language, $x=C$ for COBOL, $x=D$ for C, or $x=P$ for PL/I.

If you use CALL, specify PREPROC as the entry point name to call the translator.

In all cases, pass the following address parameters to the translator:

- The address of the translator option list
- The address of a list of DD names used by the translator (this is optional)

These addresses must be in adjacent fullwords, aligned on a fullword boundary. Register 1 must point to the first address in the list, and the high-order bit of the last address must be set to one, to indicate the end of the list. This is true for both one or two addresses.

Translator option list

The translator option list must begin on a halfword boundary. The first two bytes contain a binary count of the number of bytes in the list (excluding the count field). The remainder of the list can contain any of the translator option keywords, separated by commas, blanks, or both.

Data definition (DD name) list

The DD name list must begin on a halfword boundary. The first two bytes contain a binary count of the number of bytes in the list (excluding the count field). Each entry in the list must occupy an 8-byte field. The sequence of entries is:

Entry	Standard DD name	Entry	Standard DD name	Entry	Standard DD name
1	not applicable	3	not applicable	5	SYSIN
2	not applicable	4	not applicable	6	SYSPRINT
				7	SYSPUNCH

If you omit an applicable entry, the translator uses the standard DD name. If you use a DD name less than 8 bytes long, fill the field with blanks on the right. You can omit an entry by placing X'FF' in the first byte. You can omit entries at the end of the list entirely.

Using a CICS translator

A language translator reads your source program and creates a new one; most normal language statements remain unchanged, but CICS commands are translated into CALL statements of the form required by the language in which you are coding. The calls invoke CICS-provided “EXEC” interface modules, which later get link-edited into your load module, and these in turn invoke the requested services at execution time.

You can control the translation process by specifying translator options.

The translator options you can choose are listed in “Defining translator options” on page 74. You can specify your choices in one of two ways:

- List them as suboptions of the XOPTS option on the statement that the compiler (assembler) provides for specifying options. These statements are:

Language	Statement
COBOL	CBL
COBOL	PROCESS
C	#pragma
C++	#pragma
PL/I	* PROCESS
Assembler	*ASM or *PROCESS ¹

- List your options in the PARM operand of the EXEC job control statement for the translate step. Most installations use catalogued procedures to translate, compile (assemble) and link CICS programs, and therefore you specify this PARM field in the EXEC job control statement that invokes the procedure.

For example, if the name of the procedure for COBOL programs is DFHYITVL, and the name of the translate step within is TRN, you set translator options for a COBOL program with a statement such as this one:

```
// EXEC DFHEITCL,PARM=TRN=(VBREF,QUOTE,SPACE(2),NOCBLCARD)
```

If you specify an option by one method and the same option or an option that conflicts by the other method, the specifications in the language statement override those in the EXEC statement. Similarly, if you specify multiple values for a single option or options that conflict on either type of statement, the last setting takes precedence. Except for COBOL programs, these statements must precede each source program; there is no way to batch the processing of multiple programs in other languages.

Translator options may appear in any order, separated by one or more blanks or by a comma. If you specify them on the language statement for options, they must appear in parentheses following the XOPTS parameter, because other options are ignored by the translator and passed through to the compiler. The following COBOL example shows both translator and compiler options being passed together:

```
CBL LIB XOPTS(QUOTE SPACE(2))
```

These examples show translator options being passed alone:

```
#pragma XOPTS(FLAG(W) SOURCE);  
* PROCESS XOPTS(FLAG(W) SOURCE);  
*ASM XOPTS(NOPROLOG NOEPILOG)
```

If you use the PARM operand of the EXEC job control statement to specify options, the XOPTS keyword is unnecessary, because the only options permitted here are translator options. However, you may use XOPTS, with or without its associated parentheses. If you use XOPTS with parentheses, be sure to enclose all of the translator options. For example, the following forms are valid:

```
PARM=(op1 op2 .. opn)
PARM=(XOPTS op1 op2 .. opn)
PARM=XOPTS(op1 op2 .. opn)
```

but the following is **not** valid:

```
PARM=(XOPTS(op1 op2) opn)
```

(For compatibility with previous releases, the keyword CICS can be used as an alternative to XOPTS, except when you are translating batch EXEC DLI programs.) Remember, if you alter the default margins for C or C++ #pragma card processing using the PARM operand, the sequence margins should be altered too. You can do this using the NOSEQUENCE option.

Notes:

1. For assembler programs, *ASM statements contain translator options only. They are treated as comments by the assembler. *PROCESS statements can contain translator or assembler options for the High Level assembler, HLASM.
2. Translator and assembler options must not coexist on the same *PROCESS statement.
3. *PROCESS and *ASM statements must be at the beginning of the input and no assembler statements must appear before them. This includes comments and statements such as "PRINT ON" and "EJECT". Both *PROCESS and *ASM statements can be included, in any order.
4. *PROCESS statements containing only translator options contain information for the translator only and are not passed to the assembler
5. *PROCESS statements containing assembler options are placed in the translated program.

Defining translator options

You can specify the translator options that apply to all languages except where stated otherwise. Table 3 on page 85 lists all the translator options, the program languages that apply, and any valid abbreviations.

If your installation uses the CICS-provided procedures in the distributed form, the default options are used. These are explicitly noted in the following option descriptions. You can tell which options get used by default at your installation by looking at the SYSPRINT translator listing output from the translate step (see "The CICS-supplied translators" on page 72). If you want an option that is not the default, you must specify it, as described in "Using a CICS translator" on page 73.

Translator options

APOST

(COBOL only)

APOST indicates that literals are delineated by the apostrophe or single quote ('). QUOTE is the alternative, which indicates double quotes. The same value must be specified for the translator step and the following compile step.

#

The CICS-supplied COBOL copybooks are generated with a single quote (APOST). If you are using any CICS-supplied copybooks in your application to interface to a CICS component, ensure the APOST option is in effect, not the QUOTE option.

CBLCARD

(COBOL only) Abbreviation: CBL

CBLCARD specifies that the translator is to generate a CBL statement. This is the default—the alternative is NOCBLCARD.

CICS

CICS specifies that the translator is to process EXEC CICS commands. It is the default specification in the translator. CICS is also an old name for the XOPTS keyword for specifying translator options, which means that you can specify the CICS option explicitly either by including it in your XOPTS list or by using it in place of XOPTS to name the list. The only way to indicate that there are no CICS commands is to use the XOPTS keyword without the option CICS. You *must* do this in a batch DL/I program using EXEC DLI commands. For example, to translate a batch DL/I program written in assembler language, specify:

```
*ASM XOPTS(DLI)
```

To translate a batch program written in COBOL, containing EXEC API commands, specify:

```
CBL XOPTS(EXCI)
```


#

COBOL2

(COBOL only) Abbreviation: CO2

#

COBOL2 specifies that the translator is to generate temporary variables for use in the translated EXEC statements. In all other respects, the program is translated in the same manner as with the “COBOL3” option. COBOL2 and COBOL3 are mutually exclusive. COBOL2 is the default for COBOL.

#

Note: If you specify COBOL2 and COBOL3 by different methods, the COBOL3 option is *always* used, regardless of where the two options have been specified. If this happens, the translator issues a warning message.

COBOL3

(COBOL only) Abbreviation: CO3

COBOL3 specifies that the translator is to translate programs that are Language Environment-conforming. COBOL3 and COBOL2 are mutually exclusive. “Using the COBOL2 and COBOL3 translator options” on page 34 explains how the translator treats specific coding situations. Chapter 3, “Language Environment,” on page 9 explains what Language Environment-conforming compilers are available.

CPP

(C++ only) CPP specifies that the translator is to translate C++ programs for compilation by a supported C++ compiler, such as IBM C/C++ for MVS.

CPSM

CPSM specifies that the translator is to process EXEC CPSM commands. The alternative is NOCPSM, which is the default.

DBCS

(COBOL only)

DBCS specifies that the source program *may* contain double-byte characters. It causes the translator to treat hexadecimal codes X'0E' and X'0F' as shift-out (SO) and shift-in (SI) codes, respectively, wherever they appear in the program.

For more detailed information about how to program in COBOL using DBCS, see the section on DBCS character strings in *Enterprise COBOL for z/OS: Language Reference*.

DEBUG

(COBOL, C, C++, and PL/I only)

DEBUG instructs the translator to produce code that passes the line number through to CICS for use by the execution diagnostic facility (EDF). DEBUG is the default—NODEBUG is the alternative.

DLI

DLI specifies that the translator is to process EXEC DLI commands. You must specify it with the XOPTS option, that is, XOPTS(DLI).

EDF

EDF specifies that the execution diagnostic facility is to apply to the program. EDF is the default—the alternative is NOEDF.

EPILOG

(Assembler language only)

EPILOG specifies that the translator is to insert the macro DFHEIRET at the end of the program being translated. DFHEIRET returns control from the issuing program to the program which invoked it. If you want to use any of the options of the RETURN command, you should use RETURN and specify NOEPILOG.

EPILOG is the default—the alternative, NOEPILOG, prevents the translator inserting the macro DFHEIRET. (See the *CICS Application Programming Reference* manual for programming information about the DFHEIRET macro.)

EXCI

EXCI specifies that the translator is to process EXEC API commands for the External CICS Interface (EXCI). These commands must be used only in batch programs, and so the EXCI translator option is mutually exclusive to the CICS translator option, or any translator option that implies the CICS option. An error message is produced if both CICS and EXCI are specified, or EXCI and a translator option that implies CICS are specified.

The EXCI option is also mutually exclusive to the DLI option. EXEC API commands for the External CICS Interface cannot be coded in batch programs using EXEC DLI commands. An error message is produced if both EXCI and DLI translator commands are specified.

#

EXCI cannot be used for COBOL programs compiled with the integrated translator, but can be used with a separate translator step.

The EXCI translator option is specified by XOPTS, that is, XOPTS(EXCI).

FEPI

FEPI allows access to the FEPI API commands of the CICS Front End Programming Interface (FEPI). FEPI is only available if you have installed the CICS Front End Programming Interface. The alternative is NOFEPI. FEPI commands and design are described in the *CICS Front End Programming Interface User's Guide*.

FLAG (I, W, E, or S)

(COBOL, C, C++, and PL/I only) Abbreviation: F

FLAG specifies the minimum severity of error in the translation which requires a message to be listed.

- I** All messages.
- W** (Default) All except information messages.
- E** All except warning and information messages.
- S** Only severe and unrecoverable error messages.

GDS

(C, C++, and assembler language only)

GDS specifies that the translator is to process CICS GDS (generalized data stream) commands. For programming information about these commands, see the *CICS Application Programming Reference* manual.

GRAPHIC

(PL/I only)

GRAPHIC specifies that the source program *may* contain double-byte characters. It causes the translator to treat hexadecimal codes X'0E' and X'0F' as shift-out (SO) and shift-in (SI) codes, respectively, wherever they appear in the program.

It also prevents the translator from generating parameter lists that contain the shift-out and shift-in values in hexadecimal form. Wherever these values would ordinarily appear, the translator expresses them in binary form, so that there are no unintended DBCS delimiters in the data stream that the compiler receives.

If the compiler you are using supports DBCS, you need to prevent unintended shift-out and shift-in codes, even if you are not using double-byte characters. You can do this by specifying the GRAPHIC option for the translator, so that it does not create them, or by specifying NOGRAPHIC on the compile step, so that the compiler does not interpret them as DBCS delimiters.

For more detailed information about how to program in PL/I using DBCS, see the relevant language reference manual.

LEASM

(Assembler only)

LEASM instructs the translator to generate code for a Language Environment-conforming assembler MAIN program.

If the LEASM option is specified, the DFHEISTG, DFHEIENT, DFHEIRET and DFHEIEND macros expand differently to create a Language Environment-conforming assembler MAIN program, instead of the form of macro expansion used for assembler sub-routines in a CICS environment. This allows customer programs that have used NOPROLOG and NOEPILOG and coded their own DFHEIENT and other macros to take advantage of Language Environment support without changing their program source. For example, all programs that require more than one code base register fall into this category because the translator does not support multiple code base registers.

For an example of an assembler program translated using the LEASM option see “EXAMPLE Assembler language PROGRAM using LEASM” on page 89.

LENGTH

(COBOL, Assembler and PL/I only)

LENGTH instructs the translator to generate a default length if the LENGTH option is omitted from a CICS command in the application program. The alternative is NOLENGTH.

LINECOUNT(n)

Abbreviation: LC

LINECOUNT specifies the number of lines to be included in each page of translator listing, including heading and blank lines. The value of “n” must be an integer in the range 1 through 255; if “n” is less than 5, only the heading and one line of listing are included on each page. The default is 60.

LINKAGE

(COBOL only) Abbreviation: LIN

LINKAGE requests the translator to modify the LINKAGE SECTION and PROCEDURE DIVISION statements in top-level programs according to the existing rules.

This means that the translator will insert a USING DFHEIBLK DFHCOMMAREA statement in the PROCEDURE DIVISION, if one does not already exist, and will ensure that the LINKAGE SECTION (creating one if necessary) contains definitions for DFHEIBLK and DFHCOMMAREA.

LINKAGE is the default—the alternative is NOLINKAGE.

The LINKAGE option has no effect on the translation of classes and methods.

MARGINS(m,n[,c])

(C, C++, and PL/I only) Abbreviation: MAR

MARGINS specifies the columns of each line or record of input that contain language or CICS statements. The translator does not process data that is outside these limits, though it does include it in the source listings.

The option can also specify the position of an American National Standard printer control character to format the listing produced when the SOURCE option is specified; otherwise, the input records are listed without any intervening blank lines. The margin parameters are:

- m** Column number of left-hand margin.
- n** Column number of right-hand margin. It must be greater than m.

Note: When used as a C or C++ compiler option, the asterisk (*) is allowable for the second argument on the MARGIN option. For the translator, however, a numeric value between 1 and 100 inclusive must be specified. When the input data set has fixed-length records, the maximum value allowable for the right hand margin is 80. When the input data set has variable-length records, the maximum value allowable is 100.

- c** Column number of the American National Standard printer control character. It must be outside the values specified for m and n. A zero value for c means no printer control character. If c is nonzero, only the following printer control characters can appear in the source:

(blank)

- Skip 1 line before printing.
- 0** Skip 2 lines before printing.
- Skip 3 lines before printing.
- +** No skip before printing.
- 1** New page.

The default for C and C++ is MARGINS(1,72,0) for fixed-length records, and for variable-length records it is the same as the record length (1,record length,0). The default for PL/I is MARGINS(2,72,0) for fixed-length records, and MARGINS(10,100,0) for variable-length records.

NATLANG(EN or KA)

NATLANG specifies what language is to be used for the translator message output:

- EN** (Default) English.
- KA** Kanji.

(Take care not to confuse this option with the NATLANG API option.)

NOCBLCARD

(COBOL only)

NOCBLCARD specifies that the translator is not to generate a CBL statement. The compiler options that CICS requires are specified by the DFHYITVL procedure. You should ensure that RENT, NODYNAM, and LIB are specified..

NOCPSM

NOCPSM specifies that the translator is not to process EXEC CPSM commands. This is the default—the alternative is CPSM.

NODEBUG

(COBOL, C, C++, and PL/I only)

NODEBUG instructs the translator not to produce code that passes the line number through to CICS for use by the execution diagnostic facility (EDF).

NOEDF

NOEDF specifies that the execution diagnostic facility is not to apply to the program. There is no performance advantage in specifying NOEDF, but the option can be useful to prevent commands in well-debugged subprograms appearing on EDF displays.

NOEPILOG

(Assembler language only)

NOEPILOG instructs the translator not to insert the macro DFHEIRET at the end of the program being translated. DFHEIRET returns control from the issuing program to the program which invoked it. If you want to use any of the options of the EXEC CICS RETURN command, you should use EXEC CICS RETURN and specify NOEPILOG. NOEPILOG prevents the translator inserting the macro DFHEIRET. The alternative is EPILOG, which is the default. (See the *CICS Application Programming Reference* manual for programming information about the DFHEIRET macro.)

NOFEPI

NOFEPI disallows access to the FEPI API commands of the CICS Front End Programming Interface (FEPI). NOFEPI is the default—the alternative is FEPI.

NOLENGTH

(COBOL, Assembler and PL/I only)

NOLENGTH instructs the translator not to generate a default length if the LENGTH option is omitted from a CICS command in the application program. The default is LENGTH.

NOLINKAGE

(COBOL only)

NOLINKAGE requests the translator not to modify the LINKAGE SECTION and PROCEDURE DIVISION statements to supply missing DFHEIBLK and DFHCOMMAREA statements, or insert a definition of the EIB structure in the LINKAGE section..

This means that you can provide COBOL copybooks to define a COMMAREA and use the EXEC CICS ADDRESS command.

LINKAGE is the default.

NONUM

(COBOL only)

NONUM instructs the translator not to use the line numbers appearing in columns one through six of each line of the program as the line number in its diagnostic messages and cross-reference listing, but to generate its own line numbers. NONUM is the default—the alternative is NUM.

NOOPSEQUENCE

(C, C++, and PL/I only) Abbreviation: NOS

NOOPSEQUENCE specifies the position of the sequence field in the translator output records. The default for C and C++ is OPSEQUENCE(73,80) for fixed-length

records and NOOPSEQUENCE for variable-length records. For PL/I, the default is OPSEQUENCE(73,80) for both types of records.

NOOPTIONS

Abbreviation: NOP

NOOPTIONS instructs the translator not to include a list of the options used during this translation in its output listing.

NOPROLOG

(Assembler language only)

NOPROLOG instructs the translator not to insert the macros DFHEISTG, DFHEIEND, and DFHEIENT into the program being assembled. These macros define local program storage and execute at program entry. (See the *CICS Application Programming Reference* manual for programming information about these “prolog” macros.)

NOSEQ

(COBOL only)

NOSEQ instructs the translator not to check the sequence field of the source statements, in columns 1-6. The alternative, SEQ, is the default. If SEQ is specified and a statement is not in sequence, it is flagged.

NOSEQUENCE

(C, C++, and PL/I only) Abbreviation: NSEQ

NOSEQUENCE specifies that statements in the translator input are not sequence numbered and that the translator must assign its own line numbers.

The default for fixed-length records is SEQUENCE(73,80). For variable-length records in C and C++, the default is NOSEQUENCE and for variable-length records in PL/I the default is SEQUENCE(1,8).

NOSOURCE

NOSOURCE instructs the translator not to include a listing of the translated source program in the translator listing.

NOSPIE

NOSPIE prevents the translator from trapping irrecoverable errors; instead, a dump is produced. You should use NOSPIE only when requested to do so by the IBM support center.

NOVBREF

(COBOL, C, C++ and PL/I only)

NOVBREF instructs the translator not to include a cross-reference of commands with line numbers in the translator listing. (NOVBREF used to be called NOXREF; for compatibility, NOXREF is still accepted.) NOVBREF is the default—the alternative is VBREF.

NUM

(COBOL only)

NUM instructs the translator to use the line numbers appearing in columns one through six of each line of the program as the line number in its diagnostic messages and cross-reference listing. The alternative is NONUM, which is the default.

OPMARGINS(m,n[,c])

(C, C++ and PL/I only) Abbreviation: OM

OPMARGINS specifies the translator output margins, that is, the margins of the input to the following compiler. Normally these are the same as the input margins for the translator. For a definition of input margins and the meaning of “m”, “n”, and “c”, see MARGINS. The default for C and C++ is OPMARGINS(1,72,0) and for PL/I, the default is OPMARGINS(2,72,0).

The maximum “n” value allowable for the OPMARGINS option is 80. The output from the translator is always of a fixed-length record format.

If the OPMARGINS option is used to set the output from the translator to a certain format, it may be necessary to change the input margins for the compiler being used. If the OPMARGINS value is allowed to default this is not necessary.

OPSEQUENCE(m,n)

(C, C++, and PL/I only) Abbreviation: OS

OPSEQUENCE specifies the position of the sequence field in the translator output records. For the meaning of “m” and “n”, see SEQUENCE. The default for C and C++ is OPSEQUENCE(73,80) for fixed-length records and NOOPSEQUENCE for variable-length records. For PL/I, the default is OPSEQUENCE(73,80) for both types of records.

OPTIONS

Abbreviation: OP

OPTIONS instructs the translator to include a list of the options used during this translation in its output listing.

PROLOG

(Assembler language only)

PROLOG instructs the translator to insert the macros DFHEISTG, DFHEIEND, and DFHEIENT into the program being assembled. These macros define local program storage and execute at program entry. (See the *CICS Application Programming Reference* manual for programming information about these “prolog” macros.) PROLOG is the default—the alternative is NOPROLOG.

QUOTE

(COBOL only) Abbreviation: Q

QUOTE indicates that literals are delineated by the double quotation mark ("). The same value must be specified for the translator step and the following compiler step.

The CICS-supplied COBOL copybooks are generated with a single quote (APOST).
If you are using any CICS-supplied copybooks in your application to interface to a
CICS component, ensure the APOST option is in effect, not the QUOTE option.

SEQ (COBOL only)

SEQ instructs the translator to check the sequence field of the source statements, in columns 1-6. SEQ is the default—the alternative is NOSEQ. If a statement is not in sequence, it is flagged.

SEQUENCE(m,n) (C, C++, and PL/I only) Abbreviation: SEQ

SEQUENCE specifies that statements in the translator input are sequence numbered and the columns in each line or record that contain the sequence field. The translator uses this number as the line number in error messages and cross-reference listings. No attempt is made to sort the input lines or records into sequence. If no sequence field is specified, the translator assigns its own line numbers. The SEQUENCE parameters are:

- m** Leftmost sequence number column.
- n** Rightmost sequence number column.

The sequence number field must not exceed eight characters and must not overlap the source program (as specified in the MARGINS option).

The default for fixed-length records is SEQUENCE(73,80). For variable-length records in C and C++ the default is NOSEQUENCE and for variable-length records in PL/I the default is SEQUENCE(1,8).

SOURCE Abbreviation: S

SOURCE instructs the translator to include a listing of the translated source program in the translator listing. SOURCE is the default—the alternative is NOSOURCE.

SP
SP must be specified for application programs that contain special (SP) CICS commands or they will be rejected at translate time. These commands are ACQUIRE, COLLECT, CREATE, DISABLE, DISCARD, ENABLE, EXTRACT, INQUIRE, PERFORM, RESYNC, and SET. They are generally used by system programmers. For programming information about these commands, see the *CICS System Programming Reference* manual .

SPACE(1 or 2 or 3) (COBOL only)

SPACE indicates the type of spacing to be used in the output listing: SPACE(1) specifies single spacing, SPACE(2) double spacing, and SPACE(3) triple spacing. SPACE(3) is the default.

SPIE
SPIE specifies that the translator is to trap irrecoverable errors. SPIE is the default—the alternative is NOSPIE.

SYSEIB
SYSEIB indicates that the program is to use the system EIB instead of the application EIB. The SYSEIB option allows programs to execute CICS commands

without updating the application EIB, making that aspect of execution transparent to the application. However, this option imposes restrictions on programs using it, and should be used only in special situations. A program translated with the SYSEIB option must:

- Execute in AMODE(31), as the system EIB is assumed to be located in “TASKDATALOC(ANY)” storage.
- Obtain the address of the system EIB using the ADDRESS EIB command (if the program is translated with the SYSEIB option, this command automatically returns the address of the system EIB).
- Be aware that the use of the SYSEIB option implies the use of the NOHANDLE option on all CICS commands issued by the program. (Commands should use the RESP option as required.)

VBREF

(COBOL, C, C++, and PL/I only)

VBREF specifies whether the translator is to include a cross-reference of commands with line numbers in the translator listing. (VBREF used to be called XREF, and is still accepted.)

Translator options table

Table 3. Translator options applicable to programming language

Translator option	COBOL	C	C++	PL/I	Assembler
APOST or QUOTE	X				
CBLCARD or NOCBLCARD	X				
CICS	X	X	X	X	X
COBOL2	X				
COBOL3	X				
CPP			X		
CPSM or NOCP SM	X	X	X	X	X
DBCS	X				
DEBUG or NODEBUG	X	X	X	X	
DLI	X	X	X	X	X
EDF or NOEDF	X	X	X	X	X
EPILOG or NOEPILOG					X
EXCI	X	X	X	X	X
FEPI or NOFEPI	X	X	X	X	X
FLAG(I or W or E or S)	X	X	X	X	
GDS		X	X		X
GRAPHIC				X	
LEASM					X
LENGTH or NOLENGTH	X			X	X
LINECOUNT(n)	X	X	X	X	X
LINKAGE or NOLINKAGE	X				
MARGINS(m,n)		X	X	X	
NATLANG	X	X	X	X	X
NUM or NONUM	X				
OPMARGINS(m,n[,c])		X	X	X	
OPSEQUENCE (m,n) or NOOPSEQUENCE		X	X	X	
OPTIONS or NOOPTIONS	X	X	X	X	X
PROLOG or NOPROLOG					X
QUOTE or APOST	X				
SEQ or NOSEQ	X				
SEQUENCE (m,n) or NOSEQUENCE		X	X	X	
SOURCE or NOSOURCE		X	X	X	
SP	X	X	X	X	X
SPACE(1 or 2 or 3)	X				
SPIE or NOSPIE	X	X	X	X	X
SYSEIB	X	X	X	X	X
VBREF or NOVBREF	X	X	X	X	

Using COPY statements

The compiler (or assembler) reads the translated version of your program as input, rather than your original source. This affects what you see on your compiler (assembler) listing. It also means that COPY statements in your source code must not bring in untranslated CICS commands, because it is too late for the translator to convert them.

If you are using a separate translator and the source within any copybook contains CICS commands, you must translate it separately before translation and compilation of the program in which it will be included. If you are using the integrated CICS translator and the source within any copybook contains CICS commands, you do not have to translate it separately before compilation of the program in which it will be included.

The external program must always be passed through the CICS translator, or compiled with a compiler that has an integrated CICS translator, even if all the CICS commands are in included copybooks.

The CICS-supplied interface modules

Each of your application programs to run under CICS requires one or more interface modules (also known as **stubs**) to use the following facilities:

- The EXEC interface
- The CPI Communications facility
- The SAA Resource Recovery facility
- The CICSplex[®] SM application programming interface (for information about CICSplex SM stubs, see *CICSplex SM Application Programming Guide*).

The EXEC interface modules

Each of your CICS application programs must contain an interface to CICS. This takes the form of an EXEC interface module, used by the CICS high-level programming interface. The module, installed in the CICSTS31.CICS.SDFHLOAD library, must be link-edited with your application program to provide communication between your code and the EXEC interface program, DFHEIP.

The CPI Communications interface module

Each of your CICS application programs that uses the Common Programming Interface for Communications (CPI Communications) must contain an interface to CPI Communications. This takes the form of an interface module, used by the CICS high-level programming interface, common to all the programming languages. The module, DFHCPLC, that is installed in the CICSTS31.CICS.SDFHLOAD library, must be link-edited with each application program that uses CPI Communications.

The SAA Resource Recovery interface module

Each of your CICS application programs that uses SAA Resource Recovery must contain an interface to SAA Resource Recovery. This takes the form of an interface module, used by the CICS high-level programming interface, common to all the programming languages. The module, DFHCPLRR, that is installed in the CICSTS31.CICS.SDFHLOAD library, must be link-edited with each application program that uses the SAA Resource Recovery facility.

Using the EXEC interface modules

A language translator reads your source program and creates a new one; normal language statements remain unchanged, but CICS commands are translated into CALL statements of the form required by the language in which you are coding. The calls invoke CICS-provided “EXEC” interface modules or **stubs** , which is a function-dependent piece of code used by the CICS high-level programming interface. The stub, provided in the SDFHLOAD library, must be link-edited with your application program to provide communication between your code and the CICS EXEC interface program, DFHEIP. These stubs are invoked during execution of EXEC CICS and EXEC DLI commands.

There are stubs for each programming language.

Table 4. Interface modules

Language	Interface module name
Assembler	DFHELII and DFHEAI0
All HLL languages and Assembler MAIN programs using the LEASM option	DFHELII

The CICS-supplied stub routines work with an internal programming interface, the CICS command-level interface, which is never changed in an incompatible way. Consequently, these stub modules are upward and downward compatible, and CICS application modules never need to be re-linked to include a later level of any of these stubs.

With the exception of DFHEAI0, these stubs all provide the same function, which is to provide a linkage from EXEC CICS commands to the required CICS service. The stubs make this possible by providing various entry points that are called from the translated EXEC CICS commands, and then executing a sequence of instructions that pass control to the EXEC interface function of CICS.

DFHELII contains multiple entry points, most of which provide compatibility for very old versions of the CICS PL/I translator. It contains the entries DFHEEXEC (for C and C++ application programs), DFHEI1 (for COBOL and assembler), and DFHEI01 (for PL/1).

Each of these stubs begins with an 8 byte eyecatcher in the form DFHYxnnn, where x indicates the language supported by the stub (for example, A represents assembler, and I indicates that the stub is language independent), and nnn indicates the CICS release from which the stub was included. The letter Y in the eyecatcher indicates that the stub is read-only. Stubs supplied with very early releases of CICS contained eyecatchers in the form DFHExxxx in which the letter E denotes that the stub is not read-only.The eyecatcher for DFHELII in CICS Transaction Server for z/OS, Version 3 Release 1 is DFHYI640.

The eyecatcher can be helpful if you are trying to determine the CICS release at which a CICS application load module was most recently linked.

The DFHEAI stub must be included at the beginning of the program in the output from the link edit. To achieve this, ORDER and INCLUDE statements for DFHEAI must be in the link-edit step of your JCL. When you use the CICS-supplied assembler procedure DFHEITAL in the SDFHPROC library to translate, assemble, and link-edit application programs written in assembler language, the COPYLINK

```
# step of this procedure copies SDFHMAC(DFHEILIA). DFHEILIA contains the
# following statements that must be included:
```

```
# ORDER DFHEAI
# INCLUDE SYSLIB(DFHEAI)
```

```
# The statements are put into a temporary file that is concatenated before the
# assembled application program in the LKED step of the procedure.
```

COBOL

Each EXEC command is translated into a COBOL CALL statement that refers to the entry DFHEI1.

The following example shows the output generated by the translator when processing a simple EXEC CICS RETURN command:

```
*EXEC CICS RETURN END-EXEC
      Call 'DFHEI1' using by content x'0e0800000600001000'
end-call.
```

The reference to DFHEI1 is resolved by the inclusion of the DFHELII stub routine in the linkage editor step of the CICS-supplied procedures such as DFHYITVL or DFHZITCL.

PL/I

When translating PL/I programs each EXEC command generates a call to the entry point DFHEI01. This is done using a variable entry point DFHEI0 that is associated with the entry DFHEI01. The translator enables this by inserting the following statements near the start of each translated program:

```
DCL DFHEI0 ENTRY VARIABLE INIT(DFHEI01) AUTO;
DCL DFHEI01 ENTRY OPTIONS(INTER ASSEMBLER);
```

The translator creates a unique entry name based on DFHEI0 for each successfully translated EXEC command. The following example shows the output generated by the translator when processing a simple EXEC CICS RETURN command:

```
/* EXEC CICS RETURN TRANSID(NEXT) */
DO;
DCL DFHENTRY_B62D3C38_296F2687 BASED(ADDR(DFHEI0)) OPTIONS(INTER ASSEM
BLER) ENTRY(*,CHAR(4));
CALL DFHENTRY_B62D3C38_296F2687('xxxxxxxxxxxxxxxx' /* '0E 08 80 00 03
00 00 10 00 F0 F0 F0 F0 F0 F0 F1 F0 'X */ , NEXT);
END;
```

In the example above, DFHENTRY_B62D3C38_296F2687 is based on the entry variable DFHEI0 that is associated with the real entry DFHEI01. This technique allows the translator to create a PL/I data descriptor list for each variable entry name. The PL/I compiler can then check that variable names referenced in EXEC commands are defined with attributes that are consistent with the attributes defined by the translator in the data descriptor list. In this example, ENTRY(*,CHAR(4)) specifies that the variable (named NEXT) associated with the TRANSID option should be a character string with a length of four.

The reference to DFHEI01 is resolved by the inclusion of the DFHELII stub routine in the linkage editor step of one of the CICS-supplied procedures such as DFHYITPL.

C and C++

In a C and C++, each EXEC CICS command is translated by the command translator into a call to the function DFHEXEC. The translator enables this by inserting the following statements near the start of each translated program:

```
#pragma linkage(DFHEXEC,OS)    /* force OS linkage */
void DFHEXEC();                /* function to call CICS */
```

The following example shows the output generated by the translator when processing a simple EXEC CICS RETURN command:

```
/* EXEC CICS RETURN */
{
DFHEXEC( "\x0E\x08\x00\x2F\x00\x00\x10\x00\xF0\xF0\xF0\xF0\xF1\xF8\xF0\xF0");
}
```


#

The reference to DFHEXEC is resolved by the inclusion of the DFHELII stub routine in the linkage editor step of one of the CICS-supplied procedures such as DFHYITDL, DFHZITDL, DFHZITEL, DFHZITFL or DFHZITGL.

Assembler language

Each EXEC command is translated into an invocation of the DFHECALL macro.

The following example shows the output generated by the translator when processing a simple EXEC CICS RETURN command:

```
*          EXEC CICS RETURN
          DFHECALL =X'0E080000080001000'
```

The assembly of the DFHECALL macro invocation shown above generates code that builds a parameter list addressed by register 1, loads the address of entry DFHEI1 in register 15, and issues a BALR instruction to call the stub routine.

```
DS    0H
LA     1,DFHEITPL
LA     14,=X'0E080000080001000'
ST     14,0(,1)
OI     0(1),X'80'
L      15,=V(DFHEI1)
BALR   14,15
```

The reference to DFHEI1 is resolved by the inclusion of the DFHEAI stub routine in the linkage editor step of one of the CICS-supplied procedures such as DFHEITAL. The eyecatcher for DFHEAI in CICS Transaction Server for z/OS, Version 3 Release 1 is DFHYA640, with the release numbers indicating this stub was supplied with CICS Transaction Server for z/OS, Version 3 Release 1.

The DFHEAI0 stub for assembler application programs is included by the automatic call facility of the linkage editor or binder utility. It is called by code generated by the DFHEIENT and DFHEIRET macros to obtain and free, respectively, an assembler application program's dynamic storage area. This stub is required only in assembler application programs; there are no stubs required or supplied to provide an equivalent function for programs written in the high level languages.

EXAMPLE Assembler language PROGRAM using LEASM

Figure 11 on page 90 shows a simple CICS assembler program.

```

| *ASM XOPTS(LEASM)
| DFHEISTG DSECT
| OUTAREA DS CL200 DATA OUTPUT AREA
| *
| EIASM CSECT ,
| MVC OUTAREA(40),MSG1
| MVC OUTAREA(4),EIBTRMID
| EXEC CICS SEND TEXT FROM(OUTAREA) LENGTH(43) FREEKB ERASE
| EXEC CICS RECEIVE
| MVC OUTAREA(13),MSG2
| EXEC CICS SEND TEXT FROM(OUTAREA) LENGTH(13) FREEKB ERASE
| EXEC CICS RETURN
| *
| MSG1 DC C'xxxx: ASM program invoked. ENTER TO END.'
| MSG2 DC C'PROGRAM ENDED'
| END

```

Figure 11. a simple CICS assembler program.

When translated and assembled, it expands to Figure 12 on page 91:

```

| ASM XOPTS(LEASM)
|       DFHEIGBL ,,,LE           INSERTED BY TRANSLATOR
| *,&DFHEIDL; SETB 0 1 MEANS EXEC DLI IN PROGRAM           01-DFHEI
| *,&DFHEIDB; SETB 0 1 MEANS BATCH PROGRAM                 01-DFHEI
| *,&DFHEIRS; SETB 0 1 MEANS RSECT                         01-DFHEI
| *,&DFHEILE; SETB 1 1 MEANS LE MAIN                       01-DFHEI
| DFHEISTG DSECT
|       DFHEISTG           INSERTED BY TRANSLATOR
| *****
| *           EXEC INTERFACE DYNAMIC STORAGE           *
| *****
| DFHEISTG DSECT           EXEC INTERFACE STORAGE         @BBAC81A 01-DFHEI
|       USING *,DFHEIPLR     ESTABLISH ADDRESSABILITY     @BBAC81A 01-DFHEI
| *
| *****
| *   D Y N A M I C   S T O R A G E   A R E A   ( D S A )   *
| *****
| *
| CEEDSA DS 0D           Just keep the same label for formulae 02-CEEDS
| *
| CEEDSAFLAGS DS XL2           DSA flags                   02-CEEDS
| CEEDSALNGC EQU X'1000'       C library DSA               02-CEEDS
| CEEDSALNGP EQU X'0800'       PL/I library DSA            02-CEEDS
| CEEDSAEXIT EQU X'0008'       An Exit DSA                 02-CEEDS
| CEEDSAMEMD DS XL2           Member defined               02-CEEDS
| CEEDSABKC DS A              Addr of DSA of caller        02-CEEDS
| CEEDSAFWC DS A              Addr of DSA of last called rtn 02-CEEDS
| CEEDSAR14 DS F              Save area for register 14    02-CEEDS
| CEEDSAR15 DS F              Save area for register 15    02-CEEDS
| CEEDSAR0 DS F              Save area for register 0      02-CEEDS
| CEEDSAR1 DS F              Save area for register 1      02-CEEDS
| CEEDSAR2 DS F              Save area for register 2      02-CEEDS
| CEEDSAR3 DS F              Save area for register 3      02-CEEDS
| CEEDSAR4 DS F              Save area for register 4      02-CEEDS
| CEEDSAR5 DS F              Save area for register 5      02-CEEDS
| CEEDSAR6 DS F              Save area for register 6      02-CEEDS
| CEEDSAR7 DS F              Save area for register 7      02-CEEDS
| CEEDSAR8 DS F              Save area for register 8      02-CEEDS
| CEEDSAR9 DS F              Save area for register 9      02-CEEDS
| CEEDSAR10 DS F             Save area for register 10     02-CEEDS
| CEEDSAR11 DS F             Save area for register 11     02-CEEDS
| CEEDSAR12 DS F             Save area for register 12     02-CEEDS
| CEEDSALWS DS A              Addr of PL/I Language Working Space 02-CEEDS
| CEEDSANAB DS A              Addr of next available byte  02-CEEDS
| CEEDSAPNAB DS A              Addr of end-of-prolog NAB    02-CEEDS
|       DS 4F                                           02-CEEDS
| CEEDSATRAN DS 0A           HPL TxArea or                 02-CEEDS
| CEEDSARENT DS A           Program reentry address-IPAT   02-CEEDS
| CEEDSACILC DS A           C to Fortran ILC save area     02-CEEDS
| CEEDSAMODE DS A           Return address of module that   02-CEEDS
| *                               caused the last mode switch
|       DS 2F                                           02-CEEDS
| CEEDSARMR DS A           Addr of language specific       02-CEEDS
| *                               exception handler
| *
|       DS F                                           02-CEEDS
| CEEDSAAUTO DS 0D           Automatic storage starts here 02-CEEDS
| CEEDSAEND DS 0D           End of DSA                     02-CEEDS

```

Figure 12. the Translated assembled version (Part 1 of 8)

```

CEEDSASZ EQU   CEEDSAEND-CEEDSA      Size of DSA      02-CEEDS
CEEDSA_STDCEEDSA EQU   X'0000'      flag values of standard CEE DSA  02-CEEDS
*
*
*
DFHEISA DS      18F      SAVE AREA R14-R12 AT 12 OFF @BBAC81A 01-DFHEI
DFHEILWS DS      F      RESERVED @BBAC81A 01-DFHEI
DFHEINAB DS      F      RESERVED @BBAC81A 01-DFHEI
DFHEIRS0 DS      F      RESERVED @BBAC81A 01-DFHEI
DFHEIR13 DS      F      REGISTER 13 @BBAC81A 01-DFHEI
DFHEIRS1 DS      F      RESERVED @BBAC81A 01-DFHEI
DFHEIBP DS      F      EIB POINTER (NOT USED IF BATCH) 01-DFHEI
DFHEICAP DS      F      COMMAREA POINTER (NOT USED IF BATCH) 01-DFHEI
DFHEIV00 DS      H      HALFWORD TEMP USED BY DFHECALL 01-DFHEI
DFHEIRS2 DS      H      RESERVED @BBAC81A 01-DFHEI
DFHEIPL DS      13F      PARAMETER LIST @05C 01-DFHEI
          DS      51F      ALLOW 64 PARAMETERS FOR DLI @L2A 01-DFHEI
*
          AND IN XA2 ON, FOR EXEC CICS ALSO
DFHEIRS3 DS      F      RESERVED @L2A 01-DFHEI
DFHEIRS4 DS      F      RESERVED @L2A 01-DFHEI
DFHEITP1 DS      F      TEMPORARY POINTER 1 @L2A 01-DFHEI
DFHEITP2 DS      F      TEMPORARY POINTER 2 @L2A 01-DFHEI
DFHEITP3 DS      F      TEMPORARY POINTER 3 @L2A 01-DFHEI
DFHEITP4 DS      F      TEMPORARY POINTER 4 @L2A 01-DFHEI
*****
*          START DEFINITION OF USER DYNAMIC STORAGE          *
*****
DFHEIUSR DS      0D      ALIGN USER DYNAMIC STORAGE @BBAC81A 01-DFHEI
*
OUTAREA DS      CL200      DATA OUTPUT AREA
*
TESTLE CSECT ,
        DFHEIENT          INSERTED BY TRANSLATOR
*****
*
* CONTROL BLOCK NAME = DFHEIBLK
*
* NAME OF MATCHING PL/AS CONTROL BLOCK = None
*
* DESCRIPTIVE NAME = %PRODUCT EXEC Interface Block.
*
* @BANNER_START          02
* Licensed Materials - Property of IBM
*
* "Restricted Materials of IBM"
*
* 5697-E93
*
* (C) Copyright IBM Corp. 1990, 1993
*
*
*
* @BANNER_END
*
* STATUS = %XA20
*

```

Figure 12. the Translated assembled version (Part 2 of 8)

```

* FUNCTION = EXEC Interface Block.
*
* The exec interface block contains information on the
* transaction identifier, the time and date, and the cursor
* position on a display device. Some of the other fields are
* set indicating the next action that a program should take
* in certain circumstances.
* DFHEIBLK also contains information that will be helpful
* when a dump is being used to debug a program.
* This control block is included automatically by an
* application program using the command-level interface.
* EISEIBA in the EIS addresses the EIB.
*
*
*
* NOTES :
* DEPENDENCIES = S/370
* MODULE TYPE = Control block definition
* PROCESSOR = Assembler
*
*-----*
*
* CHANGE ACTIVITY :
* $SEG(DFHEIBLK),COMP(COMMAND),PROD(%PRODUCT) :
*
* PN= REASON REL YYMMDD HDXXIII : REMARKS
* $L1= 550 %0G 900515 HDFSPC : Add an EIB length equate
* $D1= I05119 %B1 930226 HDDHDMA : Correct comments for date field
* $P1= M60581 %B0 900116 HDAEGB : Change for PLXMAP to data areas
*
*****
* EXEC INTERFACE BLOCK
*****
DFHEIBLK DSECT EXEC INTERFACE BLOCK @BBAC81A 01-DFHEI
        USING *,DFHEIBR @BBAC81A 01-DFHEI
EIBTIME DS PL4 TIME IN 0HHMMSS FORMAT @BBAC81A 01-DFHEI
EIBDATE DS PL4 DATE IN 0CYYDDD+ FORMAT, @D1C 01-DFHEI
* where C is the century @D1A
* indicator (0=1900, 1=2000), @D1A
* YY is the year, DDD is the @D1A
* day number and '+' is the @D1A
* sign byte (positive) @D1A
EIBTRNID DS CL4 TRANSACTION IDENTIFIER @BBAC81A 01-DFHEI
EIBTASKN DS PL4 TASK NUMBER @BBAC81A 01-DFHEI
EIBTRMID DS CL4 TERMINAL IDENTIFIER @BBAC81A 01-DFHEI
EIBRSVD1 DS H RESERVED @BBAC81A 01-DFHEI
EIBCPOSN DS H CURSOR POSITION @BBAC81A 01-DFHEI
EIBCALEN DS H COMMAREA LENGTH @BBAC81A 01-DFHEI
EIBAID DS CL1 ATTENTION IDENTIFIER @BBAC81A 01-DFHEI
EIBFN DS CL2 FUNCTION CODE @BBAC81A 01-DFHEI
EIBRCODE DS CL6 RESPONSE CODE @BBAC81A 01-DFHEI
EIBDS DS CL8 DATASET NAME @BBAC81A 01-DFHEI
EIBREQID DS CL8 REQUEST IDENTIFIER @BBAC81A 01-DFHEI
EIBRSRCE DS CL8 RESOURCE NAME @BBDIA0U 01-DFHEI
EIBSYNC DS C X'FF' SYNCPOINT REQUESTED @BBDIA0U 01-DFHEI
EIBFREE DS C X'FF' FREE REQUESTED @BBDIA0U 01-DFHEI
EIBRECV DS C X'FF' RECEIVE REQUIRED @BBDIA0U 01-DFHEI

```

Figure 12. the Translated assembled version (Part 3 of 8)

```

EIBSEND DS C RESERVED @BM13417 01-DFHEI
EIBATT DS C X'FF' ATTACH RECEIVED @BBDIA0U 01-DFHEI
EIBEOC DS C X'FF' EOC RECEIVED @BBDIA0U 01-DFHEI
EIBFMH DS C X'FF' FMHS RECEIVED @BBDIA0U 01-DFHEI
EIBCOMPL DS C X'FF' DATA COMPLETE 01-DFHEI
EIBSIG DS C X'FF' SIGNAL RECEIVED 01-DFHEI
EIBCONF DS C X'FF' CONFIRM REQUESTED 01-DFHEI
EIBERR DS C X'FF' ERROR RECEIVED 01-DFHEI
EIBERRCD DS CL4 ERROR CODE RECEIVED 01-DFHEI
EIBSYNRB DS C X'FF' SYNC ROLLBACK REQ'D 01-DFHEI
EIBNODAT DS C X'FF' NO APPL DATA RECEIVED 01-DFHEI
EIBRESP DS F INTERNAL CONDITION NUMBER 01-DFHEI
EIBRESP2 DS F MORE DETAILS ON SOME RESPONSES 01-DFHEI
EIBRLDBK DS CL1 ROLLED BACK 01-DFHEI
*
EIBLENG EQU *-EIBTIME Length of EIB @L1A 01-DFHEI
*****
* END OF EXEC INTERFACE BLOCK *
*****
DFHEIBR EQU 11 EIB REGISTER @BA02936 01-DFHEI
*****
* PROLOG CODE FOR EXEC INTERFACE *
*****
*&DFHEICS; CEEENTRY PPA=DFHPPA,MAIN=YES,PLIST=OS,
* BASE=&CODEREG;,
* AUTO=(DFHEIEND-DFHEISTG)
TESTLE CSECT , 02-CEEEN
TESTLE RMODE ANY 02-CEEEN
TESTLE AMODE ANY 02-CEEEN
ENTRY TESTLE 02-CEEEN
PUSH USING 02-CEEEN
DROP , @02A 02-CEEEN
USING *,15 02-CEEEN
B CEEZ0007 02-CEEEN
DC X'00C3C5C5' 02-CEEEN
CEEY0007 DC A((((DFHEIEND-DFHEISTG)+7)/8)*8) X02-CEEEN
. Size of automatic storage.
DC A(DFHPPA-TESTLE) . Address of PPA for this program 02-CEEEN
B 1(,15) 02-CEEEN
CEEZ0007 EQU * 02-CEEEN
STM 14,12,CEEDSAR14-CEEDSA(13) 02-CEEEN
L 2,CEEINPL0007 5001D @01C 02-CEEEN
L 15,CEEINT0007 @01C 02-CEEEN
DROP 15 @01A 02-CEEEN
BALR 14,15 02-CEEEN
LR 2,1 02-CEEEN
L 14,752(,12) 02-CEEEN
OI 8(14),X'80' 02-CEEEN
BALR 3,0 @01A 02-CEEEN
USING *,3 @01A 02-CEEEN
L 3,CEE0EPV0007 @01A 02-CEEEN
POP USING @01A 02-CEEEN
USING TESTLE,3 @01A 02-CEEEN
L 1,CEEDSANAB-CEEDSA(,13) Get the current NAB 02-CEEEN
L 0,CEEY0007 02-CEEEN
ALR 0,1 Compute new value. 02-CEEEN
CL 0,CEECAAEOS-CEECAA(,12) Compare with EOS. 02-CEEEN

```

Figure 12. the Translated assembled version (Part 4 of 8)

I

```

BNH    CEEX0007                                02-CEEN
L      15,CEECAAGETS-CEECAA(,12) Get address overflow routine 02-CEEN
BALR   14,15                                Get another stack segment. 02-CEEN
LR     1,15                                02-CEEN
B      CEEX0007                                Branch around statics @01A 02-CEEN
CEEINPL0007 DC A(CEEINPL)                    @01A 02-CEEN
CEEINT0007 DC V(CEEINT)                      @01A 02-CEEN
CEE0EPV0007 DC A(TESTLE)                     @01A 02-CEEN
CEEX0007 EQU *                                02-CEEN
      ST 13,CEEDSABKC-CEEDSA(,1) Set back chain. 02-CEEN
      ST 0,CEEDSANAB-CEEDSA(,1) Set new NAB value 02-CEEN
      XC CEEDSAFLAGS-CEEDSA(,1),CEEDSAFLAGS-CEEDSA(1) . Clear 02-CEEN
      ST 1,CEEDSAFWC-CEEDSA(,13) Set forward chain. 02-CEEN
      LR 13,1                                Set save area address 02-CEEN
      USING CEEDSA,13                      Addressability to SF V1R2M0 02-CEEN
      MVC CEEDSALWS,CEECAALWS-CEECAA(12) Get LWS addr V1R2M0 02-CEEN
      LR 1,2                                02-CEEN
      BAL 1,*,+8                            @L2A 01-DFHEI
* The following gives an assembler message if DFHEISTG is too big @P7A
      DS 0S((DFHEISTG+65264-DFHEIEND-4096)/4096) @04C 01-DFHEI
      DC AL2(DFHEIEND-DFHEISTG) LENGTH OF STORAGE @L2A 01-DFHEI
      DC H'0'                                Parameter list version number @P6C 01-DFHEI
*****
* ESTABLISH DATA ADDRESSIBILITY *
*****
DFHEIPLR EQU 13                                PARAMETER LIST REGISTER @BBAC81A 01-DFHEI
      LR DFHEIPLR,15                        @BBAC81A 01-DFHEI
      USING DFHEISTG,13                    @BBAC81A 01-DFHEI
      MVC DFHEIBP(L'DFHEIBP+L'DFHEICAP),0(1) @D3AX01-DFHEI
      COPY EIB AND CA PTRS @D3A
*****
* ESTABLISH EIB ADDRESSIBILITY *
*****
      L DFHEIBR,DFHEIBP                    @BBAC81A 01-DFHEI
      USING DFHEIBLK,DFHEIBR                @BBAC81A 01-DFHEI
*****
* END OF PROLOG CODE FOR EXEC INTERFACE *
*****
      MVC OUTAREA(40),MSG1
      MVC OUTAREA(4),EIBTRMID
* EXEC CICS SEND TEXT FROM(OUTAREA) LENGTH(43) FREEKB ERASE
      DFHECALL =X'180660000800C20000082204000020',,(____RF,OUTAREA*
      ),(FB_2,=Y(43))
*****
      DS 0H                                01-DFHEC
      LA 1,DFHEIPL                        01-DFHEC
      LA 14,=X'180660000800C20000082204000020' 01-DFHEC
      SR 15,15                            01-DFHEC
      LA 0,OUTAREA                        01-DFHEC
      STM 14,0,0(1)                        01-DFHEC
      LA 14,=Y(43)                        01-DFHEC
      ST 14,12(,1)                        01-DFHEC
      OI 12(1),X'80'                      LAST ARGUMENT 01-DFHEC
      L 15,=V(DFHEI1)                    01-DFHEC
      BALR 14,15                          INVOKE EXEC INTERFACE 01-DFHEC
*****
* EXEC CICS RECEIVE

```

Figure 12. the Translated assembled version (Part 5 of 8)

```

      DFHECALL =X'040200000800000014000040000000'
*****
      DS      0H                                          01-DFHEC
      LA      1,DFHEIPL                                  01-DFHEC
      LA      14,=X'040200000800000014000040000000'    01-DFHEC
      ST      14,0(,1)                                    01-DFHEC
      OI      0(1),X'80'                                01-DFHEC
      L       15,=V(DFHEI1)                              01-DFHEC
      BALR    14,15                                INVOKE EXEC INTERFACE 01-DFHEC
*****
      MVC     OUTAREA(13),MSG2
*      EXEC CICS SEND TEXT FROM(OUTAREA) LENGTH(13) FREEKB ERASE
      DFHECALL =X'180660000800C20000082204000020',,(____RF,OUTAREA*
      ),(FB_2,=Y(13))
*****
      DS      0H                                          01-DFHEC
      LA      1,DFHEIPL                                  01-DFHEC
      LA      14,=X'180660000800C20000082204000020'    01-DFHEC
      SR      15,15                                      01-DFHEC
      LA      0,OUTAREA                                  01-DFHEC
      STM     14,0,0(1)                                  01-DFHEC
      LA      14,=Y(13)                                  01-DFHEC
      ST      14,12(,1)                                  01-DFHEC
      OI      12(1),X'80'                                01-DFHEC
      L       15,=V(DFHEI1)                              01-DFHEC
      BALR    14,15                                INVOKE EXEC INTERFACE 01-DFHEC
*****
*      EXEC CICS RETURN
      DFHECALL =X'0E0800000800001000'
*****
      DS      0H                                          01-DFHEC
      LA      1,DFHEIPL                                  01-DFHEC
      LA      14,=X'0E0800000800001000'                01-DFHEC
      ST      14,0(,1)                                    01-DFHEC
      OI      0(1),X'80'                                01-DFHEC
      L       15,=V(DFHEI1)                              01-DFHEC
      BALR    14,15                                INVOKE EXEC INTERFACE 01-DFHEC
*****
*
MSG1      DC    C'xxxx: ASM program invoked. ENTER TO END.'
MSG2      DC    C'PROGRAM ENDED'
      DFHEIRET                                INSERTED BY TRANSLATOR
*****
*      EPILOG CODE FOR EXEC INTERFACE                                *
*****
      DS      0H                                          @BBAC81A 01-DFHEI
      LA      1,CEET0014                                Get address of termination list 02-CEETE
      L       15,=V(CEETREC)                            Get address of termination rtn 02-CEETE
      BALR    14,15                                Call termination routine. 02-CEETE

CEET0014  DC    A(*+8)                                Parm 1                                02-CEETE
          DC    A(*+8+X'80000000')                    Parm 2                                02-CEETE
          DC    A(0)                                    Enc_Modifier                        02-CEETE
          DC    A(0)                                    Return code.                        02-CEETE
CEEMAIN   CSECT                                        02-CEETE
CEEMAIN   RMODE ANY                                    02-CEETE
CEEMAIN   AMODE ANY                                    02-CEETE

```

Figure 12. the Translated assembled version (Part 6 of 8)


```

        DC      A(TESTLE)                                @04A 02-CEETE
        DC      F'0'                                     02-CEETE
TESTLE   CSECT                                           02-CEETE
*****
*          END OF EPILOG CODE FOR EXEC INTERFACE          *
*****
        LTORG ,                                         @BBAC81A 01-DFHEI
          =V(DFHEI1)
          =V(CEETREC)
          =Y(43)
          =Y(13)
          =X'180660000800C20000082204000020'
          =X'040200000800000014000040000000'
          =X'0E0800000800001000'

        DS      0H                                         @F8E1S @L1C 01-DFHEI
        DFHEISTG          INSERTED BY TRANSLATOR
        DFHEIEND          INSERTED BY TRANSLATOR

*
*****
*          P R O G R A M   P R O L O G   A R E A 1   ( P P A 1 )          *
*****
*
PPA10018 DS      0F                                         02-CEEPP
DFHPPA   DS      0F                                         02-CEEPP
        DC      AL1(PPANL0018-*) Offset to the entry name length      02-CEEPP
        DC      X'CE'          LE/370 Indicator.                    02-CEEPP
        DC      B'10100000'          . PPA flags                    02-CEEPP

*          Bit 0      0 = Internal Procedure
*                   1 = External Procedure
*          Bit 1      0 = Primary Entry Point
*                   1 = Secondary Entry Point
*          Bit 2      0 = Block doesn't have a DSA
*                   1 = Block has a DSA
*          Bit 3      0 = compiled object
*                   1 = library object
*          Bit 4      0 = sampling interrupts to library
*                   1 = sampling interrupts to code
*          Bit 5      0 = not an exit DSA
*                   1 = Exit DSA
*          Bit 6      0 = own exception model
*                   1 = inherited (callers) exception model
*          Bit 7      Reserved

        DC      X'00'          Member flags                        02-CEEPP
        DC      A(PPA20018)    Addr of Compile Unit Block (PPA2)    02-CEEPP
        DC      A(0)           02-CEEPP
        DC      A(0)           Data Descriptors for this entry point 02-CEEPP
        DS      0H           02-CEEPP
PPANL0018 DC      AL2(6)      . Length of Entry Point Name          02-CEEPP
        DC      CL6'TESTLE'    . Entry Point Name                    02-CEEPP
CEEINPL   DS      0D           02-CEEPP
        DC      A(PPA2M0018)    02-CEEPP
        DC      A(CEEINPLSTST-CEEINPL) 02-CEEPP
CEEINPLSTST DS 0F           02-CEEPP
        DC      X'00'          Control Level                      @01A 02-CEEPP
        DC      X'00'          ENCLAVE=NO                         @01A 02-CEEPP
        DC      X'00'          @01A 02-CEEPP
        DC      X'07'          Number of items.                   @01C 02-CEEPP

```

Figure 12. the Translated assembled version (Part 7 of 8)

```

DC      A(PPA2M0018)      . A of A(first entry point in comp unit) 02-CEEPP
DC      V(CEESTART)       . A(Address of CEESTART)                  02-CEEPP
DC      V(CEEBETBL)       .                                     02-CEEPP
DC      A(15)              . Memeber id                             02-CEEPP
DC      A(0)               .                                     02-CEEPP
DC      XL4'00070000'      . EXECOPS(ON), PLIST                     02-CEEPP
DS      0H                                                         02-CEEPP

*
*****
*      P R O G R A M   P R O L O G   A R E A   2   ( P P A 2 )      *
*****
*
      EXTRN CEESTART                                           02-CEEPP
PPA20018 DS      0F                                           02-CEEPP
      DC      AL1(15)      Member ID                           02-CEEPP
      DC      AL1(0)       Sub ID                               02-CEEPP
      DC      AL1(0)       Member defined                       02-CEEPP
      DC      AL1(1)       Level of PPAX control blocks         02-CEEPP
PPA2S0018 DC      A(CEESTART)  A(CEESTART for this load module) 02-CEEPP
      DC      A(0)          A(Compile Debug Information (CDI) ) 02-CEEPP
      DC      A(CEETIMES-PPA20018) A(Offset to time stamp)      02-CEEPP
PPA2M0018 DC      A(TESTLE)    . A(first entry point in comp. unit) 02-CEEPP

*
*****
*      T I M E       S T A M P                                     *
*****
*
      Time Stamp
*,Time Stamp = 2004/06/17 08:51:00                               02-CEEPP
*,Version 1 Release 1 Modification 0                               02-CEEPP
CEETIMES DS      0F                                           02-CEEPP
      DC      CL4'2004'      Year                               02-CEEPP
      DC      CL2'06'        Month                              02-CEEPP
      DC      CL2'17'        Day                                02-CEEPP
      DC      CL2'08'        Hours                              02-CEEPP
      DC      CL2'51'        Minutes                            02-CEEPP
      DC      CL2'00'        Seconds                            02-CEEPP
      DC      CL2'1'         Version                            02-CEEPP
      DC      CL2'1'         Release                             02-CEEPP
      DC      CL2'0'         Modification                       02-CEEPP

*
*****
*      C O M M O N   A N C H O R   A R E A   ( C A A )             *
*****
*****
LEPTRLEN EQU      4                                           03-CEEDN
*
CEECAA      DSECT ,      CAA mapping                           02-CEECA

      (Definition of LE CAA removed)

*      TERMINATE DEFINITION OF DYNAMIC STORAGE                  *
DFHEISTG DSECT      @BBAC81A 01-DFHEI
      ORG      01-DFHEI
DFHEIEND DS      0X      END OF DYNAMIC STORAGE      @BBAC81A 01-DFHEI
      END

```

I Figure 12. the Translated assembled version (Part 8 of 8)

Chapter 9. Installing application programs

This section describes what you have to do to install an application program to run under CICS. Installation of a CICS application program involves translation and compilation of the source statements, and link-edit of the resulting object modules into CICS libraries.

An **application program** generally means any user program that uses the CICS command-level application programming interface (API). Such programs can also use:

- SQL statements
- DLI requests
- Common programming interface (CPI) statements
- SAA Resource Recovery statements
- External CICS interface commands

Note: If you are developing application programs to use the CICS dynamic transaction routing facility, use the CICS Interdependency Analyzer to detect whether the programs are likely to cause intertransaction affinity. See Chapter 15, “Affinity,” on page 221 for a description of intertransaction affinity.

This chapter includes:

- “Program installation roadmap”
- “Defining MVS residence and addressing modes” on page 100
- “Running application programs in the RDSAs” on page 103
- “Using BMS map sets in application programs” on page 105
- “Using the CICS-supplied procedures to install application programs” on page 106
- “Including the CICS-supplied interface modules” on page 108
- “Installing assembler language application programs” on page 109
- “Installing COBOL application programs” on page 110
- “Installing PL/I application programs” on page 114
- “Installing C application programs” on page 117
- “Using your own job streams” on page 120

Program installation roadmap

The following steps are required to install application programs to run under CICS. For detailed information about using the CICS-supplied procedures to install application programs, see “Using the CICS-supplied procedures to install application programs” on page 106. To use your own JCL to install application programs, see “Using your own job streams” on page 120.

1. Compile your program source if you are using a compiler with an integrated translator.
2. If your compiler does not translate CICS commands, you will need to translate the program source code, turning CICS commands into calls that are understood by the compiler, then compile or assemble the translator output to produce object code.

Notes:

- a. If your program does not use CICS commands and is only invoked by a running transaction (and never directly by CICS task initiation), no translator step is needed.

- b. CICS command-level programs that access DL/I services through either the DL/I CALL or EXEC DLI interfaces must also be translated. Applications that access DB2 services using the EXEC SQL interface need an additional precompilation step. For information about this step, see the *CICS DB2 Guide*.
3. Link-edit the object module to produce a load module, which you store in an application load library that is concatenated to the DFHRPL DD statement of the CICS startup job stream. Additional INCLUDE statements are required for applications that access DB2 services using the EXEC SQL interface. For information about these extra statements, see the *CICS DB2 Guide*.
4. Create resource definition entries, in the CSD, for any transaction that calls the program, and install them.
5. Do one of the following:
 - If you are using program autoinstall, ensure that the autoinstall user-replaceable module can correctly install a resource definition for the program.
 - If you are not using program autoinstall, create a resource definition entry in the CSD for the program, and install it.

References to the CSA or to the TCA are not allowed. You can specify YES for the system initialization parameter DISMACP to cause CICS to disable any transaction whose program invokes an obsolete CICS macro or references the CSA or the TCA.

CICS provides a utility program, DFHMSCAN, to identify the macro-level programs used by your CICS applications. For information about using the DFHMSCAN utility to identify macro-level programs, see the *CICS Operations and Utilities Guide*.

Preparing for program installation

Consider these points when installing application programs.

- If you want your application program to use CPI Communications or SAA Resource Recovery, make the appropriate interface modules available to your program. For information about the CPI Communications interface module and the SAA Resource Recovery interface module, see “The CICS-supplied interface modules” on page 86.
- If you want your application program to reside in the MVS link pack area (LPA), specify appropriate options when installing your program. Options appropriate to each language are given for the sample job streams in the following sections. For information on preparing programs to run in the link pack area (LPA), see “Running applications in the link pack area” on page 102.

For information on preparing programs to run in the read-only DSAs, see “Running application programs in the RDSAs” on page 103.

- If you want your application program to use BMS maps, first prepare the map sets. For more information, see “Using BMS map sets in application programs” on page 105.

Defining MVS residence and addressing modes

This section describes the effect of the MVS residence and addressing modes on application programs, how you can change the modes, and how you can make application programs permanently resident. An application written to run on MVS/370 can run on any MVS system, if it is link-edited with the AMODE(24) and RMODE(24) options.

A command-level program can reside above 16MB, and address areas above 16MB. The program can contain EXEC CICS, EXEC DLI, and CALL DL/I commands.

Establishing a program's addressing mode

Every program that executes in MVS is assigned two attributes: an addressing mode (AMODE), and a residency mode (RMODE). AMODE specifies the addressing mode in which your program is designed to receive control. Generally, your program is designed to execute in that mode, although you can switch modes in the program, and have different AMODE attributes for different entry points within a load module. The RMODE attribute indicates where in virtual storage your program can reside. Valid AMODE and RMODE specifications are:

AMODE(24)	Specifies 24-bit addressing mode.
AMODE(31)	Specifies 31-bit addressing mode.
AMODE(ANY)	Specifies either 24- or 31-bit addressing mode.
RMODE(24)	Indicates that the module must reside in virtual storage below 16MB. You can specify RMODE(24) for 31-bit programs that have 24-bit dependencies.
RMODE(ANY)	Indicates that the module can reside anywhere in virtual storage.

Note: C or C++ language programs must be link-edited with AMODE(31).

If you do not specify any AMODE or RMODE attributes for your program, MVS assigns the system defaults AMODE(24) and RMODE(24). To override these defaults, you can specify AMODE and RMODE in one or more of the following places. Assignments in this list overwrite assignments later in the list.

1. On the link-edit MODE control statement:
`MODE AMODE(31),RMODE(ANY)`
2. Either of the following:
 - a. In the PARM string on the EXEC statement of the link-edit job step:
`//LKED EXEC PGM=IEWL,PARM='AMODE(31),RMODE(ANY),...'`
 - b. On the LINK TSO command, which causes processing equivalent to that of the EXEC statement in the link-edit step.
3. On AMODE or RMODE statements within the source code of an assembler program. (You can also set these modes in COBOL by means of the compiler options; for information about COBOL compiler options, see the relevant application programming guide for your COBOL compiler.)

CICS address space considerations

Table 5 gives the valid combinations of the AMODE and RMODE attributes and their effects:

Table 5. Valid AMODE and RMODE specifications and their effects

AMODE	RMODE	Residence	Addressing
24	24	Below 16MB	24-bit mode
31	24	Below 16MB	31-bit mode
ANY	24	Below 16MB	31-bit mode
31	ANY	Above 16MB	31-bit mode

The following example shows link-edit control statements for a program coded to 31-bit standards:

```
//LKED.SYSIN DD *  
  MODE AMODE(31),RMODE(ANY)  
  NAME    anyname(R)      ("anyname" is your load module name)  
/*  
//
```

Making programs permanently resident

If you define a program in the CSD with the resident attribute, RESIDENT(YES), it is loaded on first reference. This applies to programs link-edited with either RMODE(ANY) or RMODE(24). However, be aware that the storage compression algorithm that CICS uses does not remove resident programs.

If there is not enough storage for a task to load a program, the task is suspended until enough storage becomes available. If any of the DSAs get close to being short on storage, CICS frees the storage occupied by programs that are not in use. (For more information about the dynamic storage areas in CICS, see the *CICS System Definition Guide*.)

Instead of making RMODE(24) programs resident, you can make them non-resident and use the library lookaside (LLA) function. The space occupied by such a program is freed when its usage count reaches zero, making more virtual storage available. LLA keeps its library directory in storage and stages (places) copies of LLA-managed library modules in a data space managed by the virtual lookaside facility (VLF). CICS locates a program module from LLA's directory in storage, rather than searching program directories on DASD. When CICS requests a staged module, LLA gets it from storage without any I/O.

Running applications in the link pack area

Programs written in assembler language, C, COBOL, or PL/I, can reside in the link pack area (LPA). To do so, they must be read-only and have been link-edited with the RENT and REFR options. Other requirements are as follows:

Assembler

Use the RENT assembler option.

C Use the RENT compiler option.

COBOL

Do not overwrite WORKING STORAGE. (The CICS translator generates a CBL statement with the required compiler RENT option (unless you specify the translator option NOCBLCARD).)

PL/I

Do not overwrite STATIC storage. (The CICS translator inserts the required REENTRANT option into the PROCEDURE statement.)

If you want CICS to use modules that you have written to these standards, and installed in the LPA, specify USELPACOPY(YES) on the program resource definitions in the CSD.

For information about installing CICS modules in the LPA, see the *CICS Transaction Server for z/OS Installation Guide*.

Running application programs in the RDSAs

Programs that are eligible to reside above 16MB, and are read-only, can reside in the CICS extended read-only DSA (ERDSA). Therefore, to be eligible for the ERDSA, programs must be:

- Properly written to read-only standards
- Written to 31-bit addressing standards
- Link-edited with the RENT attribute and the RMODE(ANY) residency attribute

Programs that are *not* eligible to reside above 16MB, and are read-only, can reside in the CICS read-only DSA (RDSA) below 16MB. Therefore, to be eligible for the RDSA, programs must be:

- Properly written to read-only standards
- Link-edited with the RENT attribute

Note: When you are running CICS with RENTPGM=PROTECT specified as a system initialization parameter, the RDSAs are allocated from key-0 read-only storage.

Programs link-edited with RENT and RMODE(ANY) are automatically loaded by CICS into the ERDSA.

ERDSA requirements for the specific languages are described as follows.

Assembler

If you want CICS to load your assembler programs in the ERDSA, assemble and link-edit them with the following options:

1. The RENT assembler option
2. The link-edit RENT attribute
3. The RMODE(ANY) residency mode

Note: If you specify these options, ensure that the program is truly read-only (that is, does not modify itself in any way—for example, by writing to static storage), otherwise storage exceptions occur. The program must also be written to 31-bit addressing standards. See the *CICS Problem Determination Guide* for some possible causes of storage protection exceptions in programs resident in the ERDSA.

The CICS-supplied procedure, DFHEITAL, has a LNKPARM parameter that specifies the XREF and LIST options only. To link-edit an ERDSA-eligible program, override LNKPARM from the calling job, specifying the RENT and RMODE(ANY) options in addition to any others you require.

For example:

```
//ASMPROG JOB 1,user_name,MSGCLASS=A,CLASS=A,NOTIFY=userid
//EITAL   EXEC DFHEITAL,
           .
           (other parameters as necessary)
           .
//        LNKPARM='LIST,XREF,RMODE(ANY),RENT'
```

Note: The CICS EXEC interface module for assembler programs (DFHEAI) specifies AMODE(ANY) and RMODE(ANY). However, because the assembler defaults your application to AMODE(24) and RMODE(24), the resulting load module also becomes AMODE(24) and RMODE(24).

If you want your application program link-edited as AMODE(31) and RMODE(ANY), you are recommended to use appropriate statements in your assembler program. For example:

```
MYPROG    CSECT
MYPROG    AMODE 31
MYPROG    RMODE ANY
```

There are two ways of setting AMODE and RMODE:

- You can set the required AMODE and RMODE specification by using link-edit (or binder) control information in the JCL PARM keyword. For example:

```
//EITAL EXEC DFHEITAL,
          LNKPARM='LIST,XREF,RENT,AMODE(31),RMODE(ANY)'
```

- Alternatively, you can use the MODE control statement in the SYSIN dataset in the link-edit, or the binder step in your JCL.

When using the binder, you may see unexpected warning messages about conflicting AMODE and RMODE specifications.

C and C/++

If you want CICS to load your C and C++ programs into the ERDSA, compile and link-edit them with:

1. The RENT compiler option.

The CICS-supplied procedures DFHYITDL or DFHYITFL (for C) and DFHYITEL or DFHYITGL (for C++) have a LNKPARM parameter that specifies a number of link-edit options. To link-edit an ERDSA-eligible program, override this parameter from the calling job, and add RENT to the other options you require. You do not need to add the RMODE(ANY) option, because the CICS EXEC interface module for C (DFHELII) is link-edited with AMODE(31) and RMODE(ANY). Therefore, your program is link-edited as AMODE(31) and RMODE(ANY) automatically when you include the CICS EXEC interface stub, see “The CICS-supplied interface modules” on page 86.

The following sample job statements show the LNKPARM parameter with the RENT option added:

```
//CPROG    JOB 1,user_name,MSGCLASS=A,CLASS=A,NOTIFY=userid
//YITDL    EXEC DFHYITDL,
          .
          (other parameters as necessary)
          .
//          LNKPARM='LIST,MAP,LET,XREF,RENT'
```

COBOL

If you use the integrated CICS translator then the compile requires the RENT compiler option, so no CBL card needs to be added during translation. COBOL programs that use a separate translation step are automatically eligible for the ERDSA, because:

- The translator option, CBLCARD (the default), causes the required compiler option, RENT, to be included automatically on the CBL statement generated by the CICS translator. If you use the translator option, NOCBLCARD, you can specify the RENT option either on the PARM statement of the compile job step, or by using the COBOL macro IGYCOPT to set installation-defined options.
- The COBOL compiler automatically generates code that conforms to read-only and 31-bit addressing standards.

- The CICS EXEC interface module for COBOL (DFHELII) is link-edited with AMODE(31) and RMODE(ANY). Therefore, your program is link-edited as AMODE(31) and RMODE(ANY) automatically when you include the CICS EXEC interface stub, see “The CICS-supplied interface modules” on page 86.

You also need to specify the reentrant attribute to link-edit. The CICS-supplied procedure, DFHYITVL, has a LNKPARM parameter that specifies a number of link-edit options. To link-edit an ERDSA-eligible program, override this parameter from the calling job, and add RENT to any other options you require. For example:

```
//COBPROG JOB 1,user_name,MSGCLASS=A,CLASS=A,NOTIFY=userid
//YITVL EXEC DFHYITVL,
          .
          (other parameters as necessary)
          .
//          LNKPARM='LIST,XREF,RENT'
```

PL/I

CICS PL/I programs are generally eligible for the ERDSA, provided they do not change static storage. The following requirements are enforced, either by CICS or PL/I:

- The required REENTRANT option is included automatically, by the CICS translator, on the PL/I PROCEDURE statement.
- The PL/I compiler automatically generates code that conforms to 31-bit addressing standards.
- The CICS EXEC interface module for PL/I (DFHELII) is link-edited with AMODE(31) and RMODE(ANY). Therefore, your program is link-edited as AMODE(31) and RMODE(ANY) automatically when you include the CICS EXEC interface stub, see “The CICS-supplied interface modules” on page 86.

You also need to specify the reentrant attribute to the link-edit. The CICS-supplied procedure, DFHYITPL, has a LNKPARM parameter that specifies a number of link-edit options. To link-edit an ERDSA-eligible program, override this parameter from the calling job, and add RENT to any other options you require. For example:

```
//PLIPROG JOB 1,user_name,MSGCLASS=A,CLASS=A,NOTIFY=userid
//YITPL EXEC DFHYITPL,
          .
          (other parameters as necessary)
          .
//          LNKPARM='LIST,XREF,RENT'
```

Note: Do not specify the RENT attribute on the link-edit step unless you have ensured the program is truly read-only (and does not, for example, write to static storage), otherwise storage exceptions will occur. See the *CICS Problem Determination Guide* for some possible causes of storage protection exceptions in programs resident in the ERDSA.

Using BMS map sets in application programs

This section describes what to do to use BMS map sets in application programs.

Before you install an application program to run under CICS:

- Create any BMS map sets used by the program, as described in Chapter 10, “Installing map sets and partition sets,” on page 125.
- Include the physical map sets (used by BMS in its formatting activities) in a library that is in the DFHRPL concatenation.

- Either include the symbolic map sets (copied into the application programs) in a user copy library, or insert them directly into the application program source.

The DFHMAPS procedure writes the symbolic map set output to the library specified on the *DSCTLIB* parameter, which defaults to the *CICSTS31.CICS.SDFHMAC* library. If you want to include symbolic map sets in a user copy library:

- Specify the library name by the *DSCTLIB=name* operand on the EXEC statement for the DFHMAPS procedure used to install physical and symbolic map sets together.
- Include a DD statement for the user copy library in the SYSLIB concatenation of the job stream used to assemble and compile the application program.

If you choose to let the DFHMAPS procedure write the symbolic map sets to the *CICSTS31.CICS.SDFHMAC* library (the default), include a DD statement for the *CICSTS31.CICS.SDFHMAC* library in the SYSLIB concatenation of the job stream used to compile the application program. This is not necessary for the DFHEITAL procedure used to assemble assembler-language programs, because these jobs already include a DD statement for the *CICSTS31.CICS.SDFHMAC* library in the SYSLIB concatenation.

- For PL/I, specify a library that has a block size of 32760 bytes. This is necessary to overcome the blocksize restriction on the PL/I compiler.

For more information about installing map sets, see Chapter 10, “Installing map sets and partition sets,” on page 125. For information about writing programs to use BMS services, see Chapter 31, “Basic mapping support,” on page 453.

Using the CICS-supplied procedures to install application programs

CICS supplies job control statements (JCL) for the translate (if required), compile, and link-edit steps, in separate cataloged procedures for each programming language supported. After CICS is installed, you should copy these procedures, installed in the *CICSTS31.CICS.SDFHPROC* library, into a procedure library.

Each procedure has a name of the form *DFHwxTyL*, where the variables *w*, *x*, and *y* depend on the type of program (EXCI batch or CICS online), the type of compiler, and the programming language. Using the preceding naming convention, the procedure names are given in Table 6 on page 107.

Table 6. Procedures for installing application programs

Language	Language Environment-conforming compilers			non-Language Environment-conforming compilers	
	Stand-alone translator	EXCI	Integrated translator	Stand-alone translator	EXCI
Assembler	-	-	-	DFHEITAL	DFHEXTAL
C	DFHYITDL (see note 1)	DFHYXTDL	DFHZITDL (see note 2)	-	-
C using the XPLINK compiler option	DFHYITFL (see note 2)	-	DFHZITFL (see note 2)	-	-
C++	DFHYITEL (see note 1)	DFHYXTEL	DFHZITEL (see note 2)	-	-
C++ using the XPLINK compiler option	DFHYITGL (see note 2)	-	DFHZITGL (see note 2)	-	-
COBOL (see note 4)	DFHYITVL	DFHYXTVL (see note 3)	DFHZITCL (see note 2)	-	-
PL/I (see note 5)	DFHYITPL (see note 2)	DFHYXTPL	DFHZITPL (see note 2)	-	-

Notes:

1. DFHYITEL may also be used for C as long as you specify the correct name of the C compiler on the COMPILER parameter.
2. The output library for the generated module is a PDSE (not a PDS).
3. A separate translator step must be used for EXCI COBOL programs as translator options are ignored when using the integrated CICS translator.
4. DFHZITCL is the recommended procedure for compiling COBOL modules, because it uses the version of the Enterprise COBOL compiler which includes the integrated CICS translator. However, if the COBOL program is intended for batch processing using the EXCI option, then the integrated translator cannot be used.
5. DFHZITPL is the recommended procedure for compiling PL/I modules as it uses the version of the Enterprise PL/I compiler which includes the integrated CICS translator. However, if the PL/I program is intended for batch processing using the EXCI option, then the integrated translator cannot be used.
6. For programs that issue EXEC DLI commands in a batch environment under Language Environment (IMS routines), use the following special procedures:

DFHYBTPL

PL/I application programs

DFHYBTVL

COBOL application programs

Installing programs in load library secondary extents

CICS supports load library secondary extents that are created while CICS is running. If you define libraries in the DFHRPL concatenation with primary and secondary extents, and secondary extents are added as a result of link-editing into the DFHRPL library while CICS is running, the CICS loader detects the occurrence, closes, and then reopens the library. This means that you can introduce new versions using the CEMT NEWCOPY command, even if the new copy of the program has caused a new library extent.

However, this can increase the search time when loading modules from the secondary extents. You should avoid using secondary extents if possible.

Note: If you are using DFHXITPL, the SYSLMOD DD statement in the binder step must refer to a PDSE (not a PDS as for the older PL/I compilers).

Including the CICS-supplied interface modules

The CICS-supplied procedures to install your online application programs in a CICS library specify the CICS library member that contains the INCLUDE statement for the appropriate language EXEC interface module. For example, the DFHYITVL procedure uses the following statements:

```
//COPYLINK EXEC PGM=IEBGENER,COND=(7,LT,COB)
//SYSUT1 DD DSN=&INDEX..SDFHSAMP(&STUB),DISP=SHR
//SYSUT2 DD DSN=&&COPYLINK,DISP=(NEW,PASS),
//          DCB=(LRECL=80,BLKSIZE=400,RECFM=FB),
//          UNIT=&WORK,SPACE=(400,(20,20))
//SYSPRINT DD SYSOUT=&OUTC
//SYSIN DD DUMMY
:
//SYSLIN DD DSN=&&COPYLINK,DISP=(OLD,DELETE)
//          DD DSN=&&LOADSET,DISP=(OLD,DELETE)
//          DD DDNAME=SYSIN
```

In this COBOL example, the symbolic parameter STUB defaults to DFHEILID. The DFHEILID member contains the statement INCLUDE SYSLIB(DFHEILII).

The supplied procedures for PL/I and C also refer to DFHEILID, which means that the DFHEILII stub is used.

If your application program is to use CPI Communications or the SAA Resource Recovery facility, do one of the following:

- Add appropriate INCLUDE statements to the LKED.SYSIN override in the job used to call the CICS-supplied procedure to install your application program. Add the following INCLUDE statements:
 - INCLUDE SYSLIB(DFHCPLC) if your program uses CPI Communications
 - INCLUDE SYSLIB(DFHCPLRR) if your program uses SAA Resource Recovery

Warning messages may appear during the link-edit step, indicating DUPLICATE definitions for the DFHEI1 entry. You may ignore these messages.

For more information about link-edit requirements, see “Using your own job streams” on page 120.

Installing assembler language application programs

You can use the DFHEITAL or DFHEXTAL procedure to translate, assemble, and link-edit application programs written in assembler language.

You can use the sample job control statements shown in Figure 13 to process application programs written in assembler language. In the procedure name, “x” depends on whether your programs are CICS application programs or EXCI batch programs. For the names of the CICS-supplied procedures, see Table 6 on page 107

```
//jobname      JOB      accounting info,name,MSGLEVEL=1
//             EXEC      PROC=DFHEXTAL 1
//TRN.SYSIN    DD      *
*ASM          XOPTS(translator options . . .) 2
              .
              assembler-language source statements
              .
/*
//LKED.SYSIN   DD      *
              NAME      anyname(R)
/*
//
```

where anyname is your load module name

Figure 13. Sample job control statements to call the DFHEXTAL procedures

Notes:

1 If you are installing a program into either of the read-only DSAs, see “Running application programs in the RDSAs” on page 103 for more details.

If you are installing a program that is to be used from the LPA, add:

- RENT to the PARM options in the EXEC statement for the ASM step of the DFHEITAL procedure
- RENT and REFR options to the LNKPARM parameter on the call to the DFHEITAL procedure

(See “Running applications in the link pack area” on page 102.)

2 For information about the translator options you can include on the XOPTS statement, see “Defining translator options” on page 74.

Figure 14 on page 110 shows the Assembler source program processed by the command level translator to produce a translator listing and an output file. This output file is then processed by the Assembler, with reference to CICS.SDFHMAC, to produce an assembler listing and a further output file. This output file is then processed by the linkage editor, with reference to CICS.SDFHLOAD to produce a linkage editor listing and a load module that is stored in CICS.SDFHLOAD.

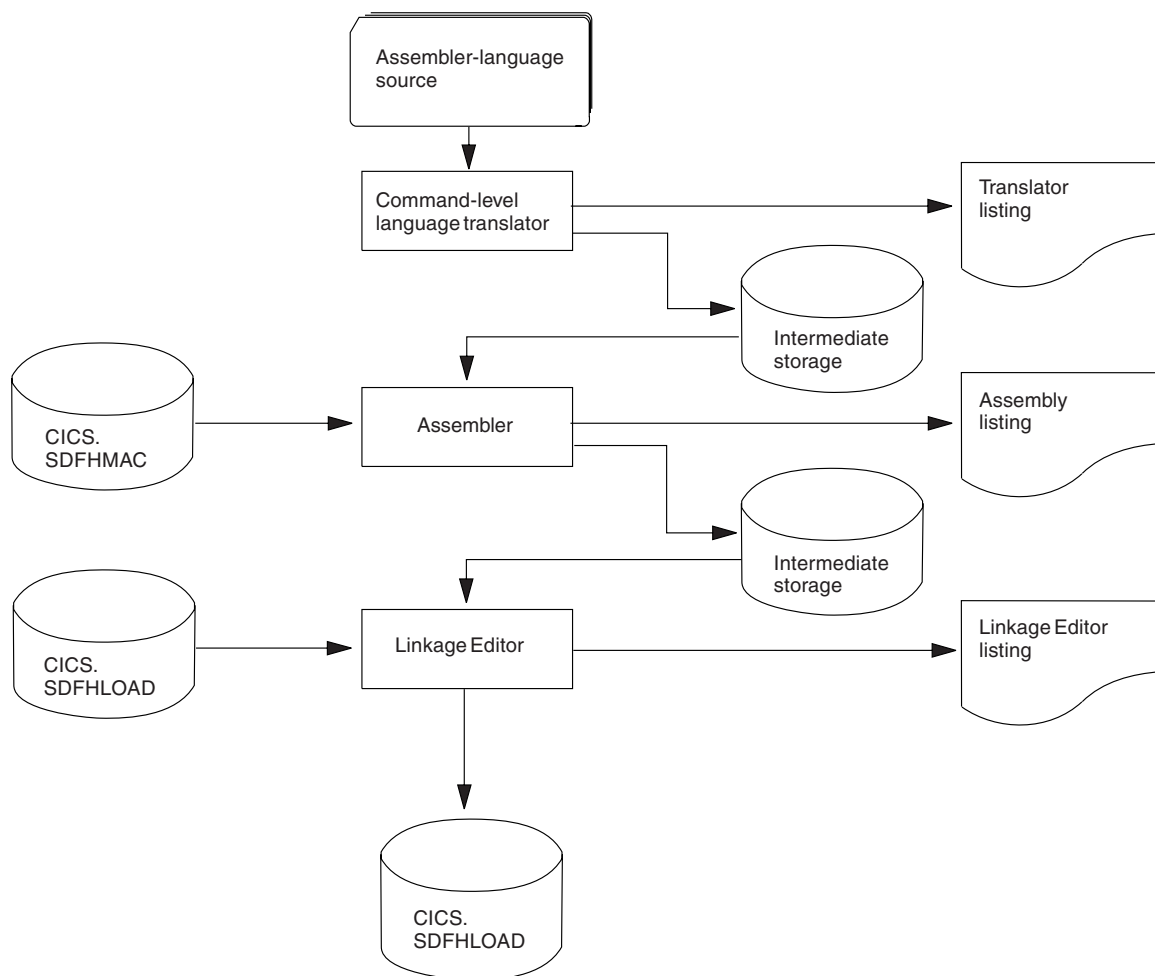


Figure 14. Installing assembler language programs using the DFHEITAL procedure

Installing COBOL application programs

Figure 15 on page 111 illustrates the flow of control in the cataloged procedures for COBOL and PL/I programs that require a separate translator step. If you use an integrated translator, there is no separate translator step. The high-level language source and CICS.SDFHLOAD both input to the compiler, and a combined translator and compiler listing is produced.

#



You can use the job control statements shown in Figure 16 on page 112 to process COBOL application programs with a separate translator. The procedure name depends on whether it is a CICS application program or an EXCI batch program. For the names of the CICS-supplied COBOL procedures, see Table 6 on page 107.

```

//jobname    JOB   accounting info,name,MSGLEVEL=1
//           EXEC  PROC=procname
//TRN.SYSIN DD      *
CBL  XOPTS(Translator options . . .)
      .
      COBOL source statements
      .
/*
//LKED.SYSIN DD      *
              NAME    anyname(R)
/*
//

```

where *procname* is the name of the procedure,
and *anyname* is your load module name.

Figure 16. Sample job control statements to call the DFHYITVL or DFHYXTVL procedures

To use the procedure DFHZITCL to invoke the integrated translator, you can use the job control statements shown in Figure 17:

```

//jobname    JOB   accounting info,name,MSGLEVEL=1
//           EXEC  DFHZITCL,PROGLIB=dsname
//COBOL.SYSIN DD *
      . COBOL source statements
      .
/*
//LKED.SYSIN DD *
              NAME anyname(R)
/*
//

```

where *anyname* is your load module name.

Figure 17. Sample job control statements to use the DFHZITCL procedure

Notes for installing COBOL programs

1 Translator options:

Specify the COBOL3 or COBOL2 translator option according to the version of the COBOL functionality required in the compile step. See “Using the COBOL2 and COBOL3 translator options” on page 34 for more information about these translator options.

Compiler options:

To compile a COBOL program, you need the compiler options RENT, NODYNAM, and LIB.

If you use the translator option, CBLCARD (the default), the CICS translator automatically generates a CBL statement containing these options. You can prevent the generation of a CBL or PROCESS card by specifying the translator option NOCBLCARD.

The PARM statement of the COB step in the CICS-supplied COBOL procedures specifies values for the compiler options. For example,


```
//COB EXEC PGM=IGYCRCTL,REGION=&REG,
// PARM='NODYNAM,LIB,OBJECT,RENT,APOST,MAP,XREF'
```

To compile a COBOL program with a compiler that has an integrated translator, you also need to use the CICS compiler option to indicate that you want the compiler to invoke the translator. The DFHZITCL procedure includes this compiler option:

```
CBLPARM='NODYNAM,LIB,MAP,CICS(''COBOL3'')'
```

Note: If you specify CICS translator options for the integrated translator in the PARM string, you need double apostrophes as shown in this example. If, however, you specify the options in your source program, you need single apostrophes (for example, you might have CBL CICS('COBOL3,SP') APOST as the CBL statement in your source program).

The CICS-supplied COBOL procedures do not specify values for the SIZE and BUF options. The defaults are SIZE=MAX, and BUF=4K. SIZE defines the amount of virtual storage available to the compiler, and BUF defines the amount of dynamic storage to be allocated to each compiler buffer work file. You can change these options with a PARM.COB parameter in the EXEC statement that invokes the procedure. For example:

```
EXEC PROC=procname,PARM.COB='SIZE=512K,BUF=16K,.,.,.'
```

You can change compiler options by using any of the following methods:

- By overriding the PARM statement defined on the COB step of the CICS-supplied COBOL procedures.
If you specify a PARM statement in the job that calls the procedure, it overrides **all** the options specified in the procedure JCL. Ensure that all the options you want are specified in the override, or in a CBL statement.
- Specifying a CBL statement at the start of the source statements in the job stream used to call the CICS-supplied COBOL procedures.
- The COBOL installation defaults macro, IGYCOPT. This is needed if you do not use a CBL statement (that is, you have specified the translator option NOCBLCARD).

For information about the translator option CBLCARDINOCBLCARD, see “Defining translator options” on page 74. If you choose to use the NOCBLCARD option, also specify the COBOL compiler option ALLOWCBL=NO to prevent an error message of IGYOS4006-E being issued. For more information about the ALLOWCBL compiler option, see the relevant *Installation and Customization* manual for your version of COBOL.

2 If you have no input for the translator, you can specify DD DUMMY instead of DD *. However, if you specify DD DUMMY, also code a suitable DCB operand. (The translator does not supply all the data control block information for the SYSIN data set.)

3 If the stand-alone translator supplied with CICS TS is used, the translator options on the XOPTS statement override similar options in the CICS-supplied COBOL procedures.

For information about the translator options you can include on the XOPTS statement, see “Defining translator options” on page 74.

When the integrated CICS translator is used, the COBOL compiler recognizes only the keyword CICS for defining translator options, not XOPTS.

4 You can ignore weak external references unresolved by the link-edit.

The link-edit job step requires access to the libraries containing the environment-specific modules for CICS, and the Language Environment link-edit modules, as appropriate. Override or change the names of these libraries if the modules and library subroutines are installed in libraries with different names.

If you are installing a program into either of the read-only DSAs, see “Running application programs in the RDSAs” on page 103 for more details.

If you are installing a program that is to be used from the LPA, add the RENT and REFR options to the LNKPARM parameter on the call to the CICS-supplied COBOL procedures. (See “Running applications in the link pack area” on page 102.)

Installing PL/I application programs

Figure 15 on page 111 illustrates the flow of control in the cataloged procedures for PL/I programs.

For more information about preparing PL/I programs, see the *PL/I Programming Guide*.

Sample JCL to install PL/I application programs

You can use the job control statements shown in Figure 18 to process PL/I application programs with a separate translator.

In the procedure name, the value of “x” depends on whether it is a CICS application program or an EXCI batch program. For the names of the CICS-supplied procedures, see Table 6 on page 107.

```
//jobname JOB accounting info,name,MSGLEVEL=1
// EXEC PROC=DFHYxTPL
//TRN.SYSIN DD *
*PROCESS XOPTS(translator options...)PL/I compiler options...;
          .
          PL/I source statements
          .
/*
//LKED.SYSIN DD *
          NAME anyname(R)
/*
//
```

where anyname is your load module name

Figure 18. Sample job control statements to call the DFHYxTPL procedures

Notes for installing a PL/I program:

1 The PL/I COUNT runtime option is not supported by Language Environment. The REPORT option is replaced by the RPTSTG and RPTUPTS Language Environment options. See the *z/OS Language Environment Migration Guide*.

2 If you have no input for the translator, you can specify DD DUMMY instead of DD *. However, if you specify DD DUMMY also code a suitable DCB operand. (The translator does not supply all the data control block information for the SYSIN data set.)

3 Translator and compiler options:

For information about the translator options you can include on the XOPTS statement, see “Defining translator options” on page 74.

Ignore the message from the PL/I compiler: “IEL0548I PARAMETER TO MAIN PROCEDURE NOT VARYING CHARACTER STRING”.

Warning messages may appear from the PL/I compiler stating that arguments and parameters do not match for calls to procedure DFHxxxx. These messages indicate that arguments specified in operands to CICS commands may not have the correct data type. Carefully check all fields mentioned in these messages, especially **receiver** fields.

4 If you include the CALL PLIDUMP statement in an application program, output goes to the CESE transient data destination. The CICS supplied resource definition group, in the CSD, DFHDCTG, contains an entry for CESE.

5 Link-edit considerations:

You can ignore weak external references unresolved by the link-edit.

If you are installing a program into either of the read-only DSAs, see “Running application programs in the RDSAs” on page 103 for more details.

If you are installing a program that is to be used from the LPA, add the RENT and REFR options to the LNKPARM parameter on the call to the DFHYxTPL procedure. (See “Running applications in the link pack area” on page 102 for more information.)

PL/I procedure with an integrated translator

To use the new procedure DFHZITPL to invoke the integrated translator, you can use the following sample JCL:

```
//jobname JOB accounting info,name,MSGLEVEL=1
// EXEC DFHZITPL,PROGLIB=dsname 1
//PLI.SYSIN DD *
    .
    . PLI source statements
    .
/*
//LKED.SYSIN DD *
    NAME anyname(R)
/*
//
```

where anyname is your load module name.

Figure 19. Sample job control statements to use the DFHZITPL procedure

Notes for installing PLI programs with an integrated translator

1 Translator options:

The DFHZITPL procedure includes the following compiler options to indicate that you want the compiler to invoke the translator:

```
PLIPARM=('SOURCE,OPTIONS,SYSTEM(CICS),PP(CICS)')
```

Note: In this procedure, the SYSLMOD DD statement in the LKED step must refer to a PDSE (not a PDS as for the older PL/I compilers).

Installing C application programs

Figure 20 shows the flow of control in the DFHYxTzL cataloged procedures for C command-level programs.

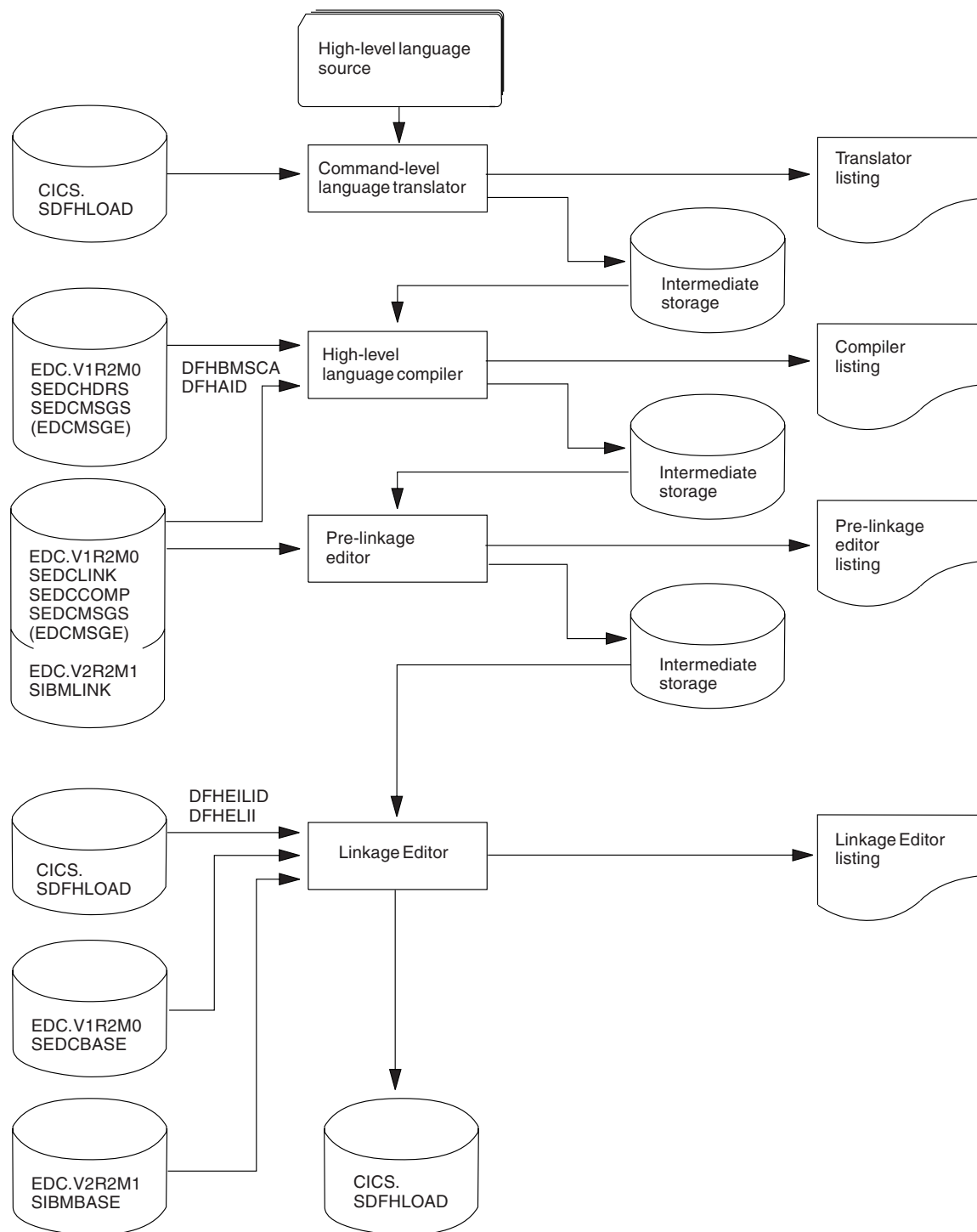


Figure 20. Installing C programs using the DFHYxTzL procedure

There are translator, compiler, pre-linkage editor and linkage editor steps, each producing a listing and an intermediate file that is passed to the next step. C libraries are referenced in the compiler, pre linkage editor and linkage editor steps.

#

Note: When you choose the XPLINK compiler option, there is no pre-link step in the diagram above.

Before you can install any C programs, you must have installed the C library and compiler and generated CICS support for C. (See the *CICS Transaction Server for z/OS Installation Guide*.)

Sample JCL to install C application programs

You can use the job control statements shown in Figure 21 to process C application programs. In the procedure name, x depends on whether your program is a CICS application program or an EXCI batch program. For the names of the CICS-supplied procedures, see Table 6 on page 107.

```
//jobname      JOB   accounting info,name,MSGLEVEL=1
//              EXEC  PROC=DFHYxTzL
//TRN.SYSIN    DD      *
#pragma XOPTS(Translator options . . .)
               .
               C source statements
               .
/*
//LKED.SYSIN    DD      *
                   NAME      anyname(R)
/*
//
```

1
2
3

4

where anyname is your load module name

Figure 21. Sample JCL to call the DFHYxTzL procedures

Notes for installing a C program:

1 Compiler options:

You can code compiler options by using the parameter override (PARM.C) in the EXEC statement that invokes the procedure, or on a ~pragma options directive.

2 If you have no input for the translator, you can specify DD DUMMY instead of DD *. However, if you specify DD DUMMY, also code a suitable DCB operand. (The translator does not supply all the data control block information for the SYSIN data set.)

3 Translator options: For information about the translator options you can include on the XOPTS statement, see “Defining translator options” on page 74 .

4 If you are installing a program into either of the read-only DSAs, see “Running application programs in the RDSAs” on page 103 for more details.

If you are installing a program that is to be used from the LPA, add the RENT and REFR options to the LNKPARM parameter on the call to the DFHYxTzL procedure. (See “Running applications in the link pack area” on page 102 for more information.)

C language programs must be link-edited with AMODE(31), so the DFHYxTzL procedures specify AMODE(31) by default.

To use the procedures to invoke the integrated translator for XL C, you can use the job control statements shown in Figure 22:

```
//jobname    JOB   accounting info,name,MSGLEVEL=1
//           EXEC  DFHZITxL,PROGLIB=dsnnname
//C.SYSIN    DD *
//           .
//           . C source statements
//           .
//LKED.SYSIN  DD *
//           NAME  anyone(R)
//
```

1

where anyone is your load module name.

Figure 22. Sample job control statements to invoke the integrated translator for XL C

1 Translator name:

Specify DFHZITDL for C programs without XPLINK or DFHZITFL for C programs with XPLINK.

To use the procedures to invoke the integrated translator for XL C++, you can use the job control statements shown in Figure 23:

```
//jobname    JOB   accounting info,name,MSGLEVEL=1
//           EXEC  DFHZITxL,PROGLIB=dsnnname
//CPP.SYSIN   DD *
//           .
//           . C++ source statements
//           .
//LKED.SYSIN  DD *
//           NAME  anyone(R)
//
```

1

where anyone is your load module name.

Figure 23. Sample job control statements to invoke the integrated translator for XL C++

1 Translator name:

Specify DFHZITEL for C++ programs without XPLINK, or DFHZITGL for C++ programs with XPLINK.

Including pre-translated code with your C source code

The translator may generate dfhexec or DFHEXEC. If both versions are present in your program, you will see error message IEW2456E. There are two ways to prevent this error.

1. Recompile the old code containing dfhexec.
2. Use the prelinker RENAME control statement in the job, as shown below.

```

//jobname    JOB   accounting info,name,MSGLEVEL=1
//           EXEC  PROC=DFHYxTzL
//TRN.SYSLIN DD      *
#pragma XOPTS(Translator options . . .)

      .
      C source statements
      .
/*
//PLKED.SYSLIN DD      *
      RENAME dfhexec DFHEI1
//LKED.SYSLIN DD      *
      NAME  anyone(R)
/*
//

```

where anyone is your load module name

Figure 24. Sample JCL to rename dfhexec

Using your own job streams

If you want to write your own JCL to translate, assemble (or compile), and link-edit your application programs, you can use the supplied cataloged procedures as a model. They are installed in the CICS31.CICS.SDFHPROC library.

The rest of this section summarizes the important points about the translator and each of the main categories of program. For simplicity, the following discussion states that you load programs into CICS31.CICS.SDFHLOAD or IMS™.PGMLIB. In fact, you can use any libraries, but only when they are either included in the DFHRPL library concatenation in the CICS job stream, or included in the STEPLIB library concatenation in the batch job stream (for a stand-alone IMS batch program).

Note: The IMS libraries referred to in the job streams are identified by IMS.libnam (for example IMS.PGMLIB). If you use your own naming convention for IMS libraries, please rename the IMS libraries accordingly.

Translator requirements

The CICS translator requires a minimum of 256KB of virtual storage. You may need to use the translator options CICS and DLI.

Online programs that use EXEC CICS or EXEC DLI commands

1. Always use the translator option CICS. If the program issues EXEC DLI commands, use the translator option DLI.
2. The link-edit input (defined by the SYSLIN DD statement) must include the correct interface module **before** the object deck. Therefore, place an INCLUDE statement for the interface module before the object deck. Also put ORDER statements before the INCLUDE statements, and an ENTRY statement after all the INCLUDE statements.

The interface modules are:

DFHEAI

Assembler

DFHELII

All HLL languages

In the CICS-supplied procedures, the input to the link-edit step (defined by the SYSLIN DD statement) concatenates a library member with the object deck. This member contains an INCLUDE statement for the required interface module. For example, the DFHYITVL procedure concatenates the library member DFHEILID, which contains the following INCLUDE statement:

```
INCLUDE SYSLIB(DFHEILII)
```

3. Place the load module output from the link-edit (defined by the SYSLMOD DD statement) in CICSTS31.CICS.SDFHLOAD, or your own program library.

Figure 25 shows sample JCL and an inline procedure, based on the CICS-supplied procedure DFHYITVL, that can be used to install COBOL application programs. The procedure does not include the COPYLINK step and concatenation of the library member DFHEILID that contains the INCLUDE statement for the required interface module (as included in the DFHYITVL procedure). Instead, the JCL provides the following INCLUDE statement:

```
INCLUDE SYSLIB(DFHEILII)
```

If this statement was not provided, the link-edit would return an error message for unresolved external references, and the program output would be marked as not executable.

```

/*      The following JCL could be used to execute this procedure
/*
//APPLPROG EXEC MYITVL,
//      INDEX='CICSTS31.CICS
//      PROGLIB='CICSTS31.CICS.SDFHLOAD',
//      DSCTLIB='CICSTS31.CICS.SDFHCOB',
//      INDEX2='user.qualif'
//      OUTC=A,                      Class for print output
//      REG=4M,                      Region size for all steps
//      LNKPARM='LIST,XREF',         Link edit parameters
//      WORK=SYSDA                   Unit for work datasets

//TRN.SYSIN DD *
/*      .
/*      . Application program
/*      .
/*
//LKED.SYSIN DD *
//      INCLUDE SYSLIB(DFHEILII)
//      NAME anyname(R)

/*
//MYITVL PROC SUFFIX=1$,             Suffix for translator module
//      INDEX='CICSTS31.CICS', Qualifier(s) for CICS libraries
//      PROGLIB='CICSTS31.CICS.SDFHLOAD', Name of o/p library
//      DSCTLIB='CICSTS31.CICS.SDFHCOB', Private macro/dsect
//      AD370HLQ='SYS1',             Qualifier(s) for AD/Cycle compiler
//      LE370HLQ='SYS1',             Qualifier(s) for Language Environment libraries
//      OUTC=A,                      Class for print output
//      REG=4M,                      Region size for all steps
//      LNKPARM='LIST,XREF',         Link edit parameters
//      WORK=SYSDA                   Unit for work datasets
//

```

Figure 25. Sample user-defined JCL to install a COBOL program (Part 1 of 2)

```

/*      This procedure contains 3 steps
/*      1.  Exec the COBOL translator (using the supplied suffix 1$)
/*      2.  Exec the COBOL compiler
/*      3.  Linkedit the output into dataset &PROGLIB

//TRN   EXEC PGM=DFHECP &SUFFIX,,
//      PARM='COBOL3',
//      REGION=&REG

//STEPLIB DD DSN=&INDEX..SDFHLOAD,DISP=SHR
//SYSPRINT DD SYSOUT=&OUTC
//SYSPUNCH DD DSN=&&SYSCIN,
//      DISP=(,PASS),UNIT=&WORK,
//      DCB=BLKSIZE=400,
//      SPACE=(400,(400,100))
/*
//COB   EXEC PGM=IGYCRCTL,REGION=&REG,
//      PARM='NODYNAM,LIB,OBJECT,RENT,APOST,MAP,XREF'
//STEPLIB DD DSN=&AD370HLQ..SIGYCOMP,DISP=SHR
//SYSLIB DD DSN=&DSCTLIB,DISP=SHR
//      DD DSN=&INDEX..SDFHCOB,DISP=SHR
//      DD DSN=&INDEX..SDFHMAC,DISP=SHR
//      DD DSN=&INDEX..SDFHSAMP,DISP=SHR
//SYSPRINT DD SYSOUT=&OUTC
//SYSIN DD DSN=&&SYSCIN,DISP=(OLD,DELETE)
//SYSLIN DD DSN=&&LOADSET,DISP=(MOD,PASS),
//      UNIT=&WORK,SPACE=(80,(250,100))
//SYSUT1 DD UNIT=&WORK,SPACE=(460,(350,100))
//SYSUT2 DD UNIT=&WORK,SPACE=(460,(350,100))
//SYSUT3 DD UNIT=&WORK,SPACE=(460,(350,100))
//SYSUT4 DD UNIT=&WORK,SPACE=(460,(350,100))
//SYSUT5 DD UNIT=&WORK,SPACE=(460,(350,100))
//SYSUT6 DD UNIT=&WORK,SPACE=(460,(350,100))
/*
//LKED EXEC PGM=IEWL,REGION=&REG,
//      PARM='&LNKPARM',COND=(5,LT,COB)
//SYSLIB DD DSN=&INDEX..SDFHLOAD,DISP=SHR
//      DD DSN=&LE370HLQ..SCEELKED,DISP=SHR
//SYSLMOD DD DSN=&PROGLIB,DISP=SHR
//SYSUT1 DD UNIT=&WORK,DCB=BLKSIZE=1024,
//      SPACE=(1024,(200,20))
//SYSPRINT DD SYSOUT=&OUTC
//SYSLIN DD DSN=&&COPYLINK,DISP=(OLD,DELETE)
//      DD DSN=&&LOADSET,DISP=(OLD,DELETE)
//      DD DDNAME=SYSIN
//PEND
//*

```

Figure 25. Sample user-defined JCL to install a COBOL program (Part 2 of 2)

Online programs that use the CALL DLI interface

1. Specify the translator option CICS, but not the translator option DLI.

Note: For a program that does not use CICS commands and is only invoked by a running transaction (and never directly by CICS task initiation), no translator step is needed.

2. The interface module, DFHDLIAI, is automatically included by the link-edit. If you use an INCLUDE statement in the link-edit input, place it **after** the object deck.
3. Include copybook DLIUIB in your program.

#

4. Place the load module output from the link-edit (defined by the SYSLMOD DD statement) in CICSTS31.CICS.SDFHLOAD, or a user-defined application program library.

Batch or BMP programs that use EXEC DLI commands

1. The translator option DLI is required. Do not specify the translator option CICS.
2. The INCLUDE statement for the interface module must **follow** the object deck in the input to the link-edit (defined by the SYSLIN DD statement). The interface module, DFSLI000, which resides on IMS.RESLIB, is the same for all programming languages. If you include CICSTS31.CICS.SDFHLOAD in the input to the link-edit (defined by the SYSLIB DD statement), concatenate it **after** IMS.RESLIB.
3. Place the load module output from the link-edit (defined by the SYSLMOD DD statement) in IMS.PGMLIB, or a library concatenated in the STEPLIB DD statement of the batch job stream.

Batch or BMP programs that use DL/I CALL commands

If you want to prepare assembler, COBOL, or PL/I programs that use the DL/I CALL interface, do not use any of the CICS-supplied procedures. Programs that contain CALL ASMTDLI, CALL CBLTDLI, or CALL PLITDLI should be assembled or compiled, and link-edited, as IMS applications, and are not subject to any CICS requirements. See the relevant IMS manual for information about how to prepare application programs that use the DL/I CALL interface.

Chapter 10. Installing map sets and partition sets

This chapter describes how to assemble and link-edit map sets and partition sets for use with the basic mapping (BMS) facility of CICS. It also describes how to install HTML templates generated from BMS maps. See “Using the DFHMAPT procedure to install HTML templates from BMS maps” on page 131 and the *CICS Internet Guide* for information about using HTML templates.

If your program uses BMS maps, you need to create the maps. The traditional method for doing this is to code the map in BMS macros and assemble these macros. You actually do the assembly twice, with different output options.

- One assembly creates a set of definitions. You copy these definitions into your program using the appropriate language statement, and they allow you to refer to the fields in the map by name.
- The second assembly creates an object module that is used when your program actually executes.

The process is illustrated in the following diagram:.

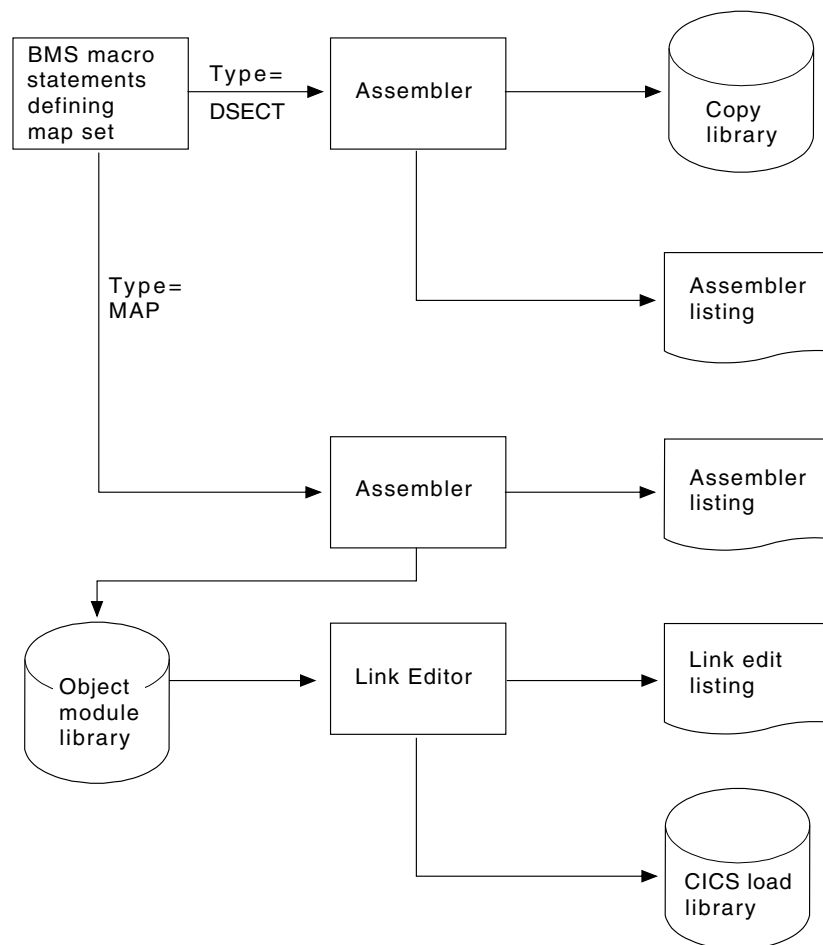


Figure 26. Preparing a map

Whatever way you produce maps, you need to create a map before you compile (assemble) any program that uses it. In addition, if you change the map, you

usually need to recompile (reassemble) all programs that use it. Some changes affect only the physical map and are not reflected in the corresponding symbolic map used by the program. One of these is a change in field position that does not alter the order of the fields. However, changes in data type, field length, field sequence, and others do affect the symbolic map, and it is always safest to recompile (reassemble).

CICS also supports the definition of BMS map sets and partition sets interactively by using licensed programs such as the IBM Screen Definition Facility II (SDF II), program number 5665-366. For more information about SDF II, see the *Screen Definition Facility II Primer for CICS/BMS Programs* and *Screen Definition Facility II General Information* manuals.

For information about writing programs to use BMS services, see Chapter 31, “Basic mapping support,” on page 453.

CICS loads BMS map sets and partition sets above the 16MB line if you specify the residency mode for the map set or partition set as RMODE(ANY) in the link-edit step. If you are using either map sets or partition sets from earlier releases of CICS, you can load them above the 16MB line by link-editing them again with RMODE(ANY). For examples of link-edit steps specifying RMODE(ANY), see the sample job streams in this chapter.

This chapter includes:

- “Installing map sets”
- “Installing partition sets” on page 133
- “Defining programs, map sets, and partition sets to CICS” on page 134

Installing map sets

This section first describes the types of map sets, how you define them, and how CICS recognizes them. This is followed by a description of how to prepare physical map sets and symbolic description map sets separately. Finally, there is a description of how to prepare both physical and symbolic description map sets in one job. In these descriptions, it is assumed that the SYSPARM parameter is used to distinguish the two types of map sets.

Types of map sets

To install a map set, you must actually prepare two types of map sets:

- A **physical** map set, used by BMS to translate data from the standard device-independent form used by application programs to the device-dependent form required by terminals.
- A **symbolic description** map set, used in the application program to define the standard device-independent form of the user data. This is a DSECT in assembler language, a data definition in COBOL, a BASED or AUTOMATIC structure in PL/I, and a “struct” in C/370.

Physical map sets must be cataloged in the CICS load library. Symbolic description map sets can be cataloged in a user copy library, or inserted directly into the application program itself.

The map set definition macros are assembled twice; once to produce the physical map set used by BMS in its formatting activities, and once to produce the symbolic description map set that is copied into the application program.

Defining the type of map set you require

The two types of map set can be distinguished by either:

- The TYPE operand of the DFHMSD macro
- Use of the SYSPARM operand on the EXEC statement of the job used to assemble the map set

If you use the SYSPARM operand for this purpose, the TYPE operand of the DFHMSD macro is ignored. Using SYSPARM allows both the physical map set and the symbolic description map set to be generated from the same unchanged set of BMS map set definition macros.

Map sets can be assembled as either **unaligned** or **aligned** (an aligned map is one in which the length field is aligned on a halfword boundary). Use unaligned maps except in cases where an application package needs to use aligned maps.

The SYSPARM value alone determines whether the map set is aligned or unaligned, and is specified on the EXEC PROC=DFHMAPS statement. The SYSPARM operand can also be used to specify whether a physical map set or a symbolic description map set (DSECT) is to be assembled, in which case it overrides the TYPE operand. If neither operand is specified, an unaligned DSECT is generated.

The TYPE operand of the DFHMSD macro can only define whether a physical or symbolic description map set is required.

For the possible combinations of operands to generate the various types of map set, see Table 7.

Table 7. SYSPARM and DFHMSD operand combinations for map assembly

Type of map set	SYSPARM operand of EXEC DFHMAPS statement	TYPE operand of DFHMSD macro
Aligned symbolic description map set (DSECT)	A A ADSECT	Not specified DSECT Any (takes SYSPARM)
Aligned physical map set	A AMAP	MAP Any (takes SYSPARM)
Unaligned symbolic description map set (DSECT)	Not specified Not specified DSECT	Not specified DSECT Any (takes SYSPARM)
Unaligned physical map set	Not specified MAP	MAP Any (takes SYSPARM)

The physical map set indicates whether it was assembled for aligned or unaligned maps. This information is tested at execution time, and the appropriate map alignment used. Thus aligned and unaligned map sets can be mixed.

Using extended data stream terminals

Applications and maps designed for the 3270 Information Display System run unchanged on devices supporting extensions to the 3270 data stream such as color, extended highlighting, programmed symbols, and validation. To use fixed extended attributes such as color, you only need to reassemble the physical map set. If dynamic attribute modification by the application program is needed, you

must reassemble both the physical and symbolic description map sets, and you must reassemble or recompile the application program.

Installing physical map sets

Figure 27 shows the the assembler and linkage editor steps for installing physical map sets.

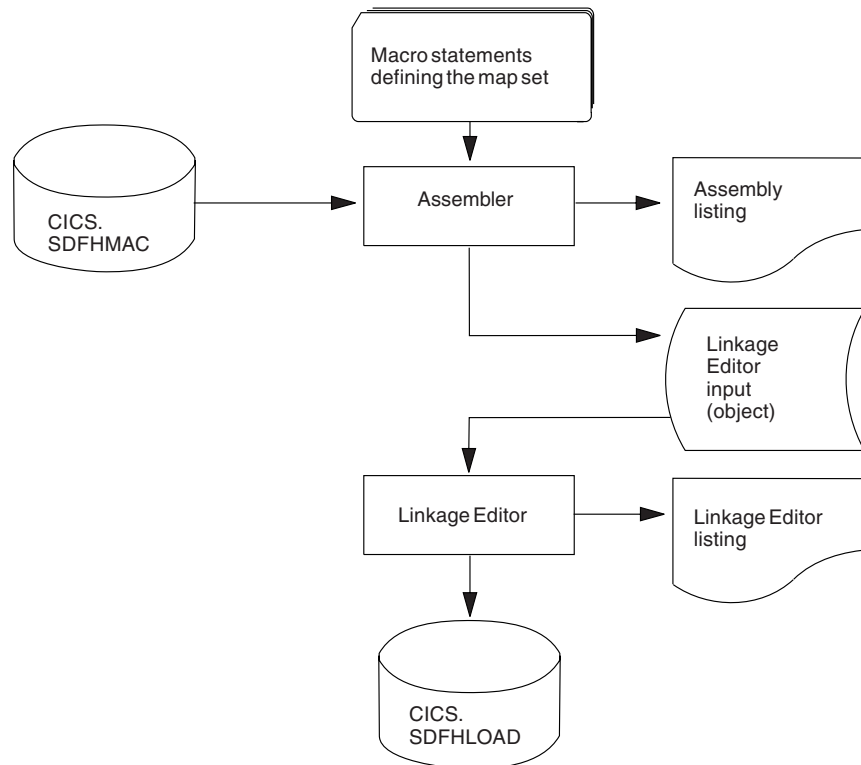


Figure 27. Installing physical map sets

Figure 28 gives an example job stream for the assembly and link-editing of physical map sets.

```

//PREP      JOB 'accounting information',CLASS=A,MSGLEVEL=1
//STEP1     EXEC PROC=DFHASMVS,PARM=ASSEM='SYSPARM(MAP)'
//SYSPUNCH DD DSN=&&TEMP,DCB=(RECFM=FB,BLKSIZE=2960),
//          SPACE=(2960,(10,10)),UNIT=SYSDA,DISP=(NEW,PASS)
//SYSIN     DD *
:
:          Macro statements defining the map set
:
/*
//STEP2     EXEC PROC=DFHLNKVS,PARM='LIST,LET,XREF'
//SYSLIN    DD DSN=&&TEMP,DISP=(OLD,DELETE)
//          DD *
//          MODE RMODE(ANY|24)
//          NAME mapsetname(R)
/*
//
  
```

Figure 28. Assembling and link-editing a physical map set

Notes:

1 For halfword-aligned length fields, specify the option SYSPARM(AMAP) instead of SYSPARM(MAP).

2 Physical map sets are loaded into CICS-key storage, unless they are link-edited with the RMODE(ANY) and RENT options. If they are link-edited with these options, they are loaded into key-0 protected storage, provided that RENTPGM=PROTECT is specified on the RENTPGM initialization parameter. However, it is recommended that map sets (except for those that are only sent to 3270 or LU1 devices) should not be link-edited with the RENT or the REFR options because, in some cases, CICS modifies the map set. Generally, use the RENT or REFR options for map sets that are only sent to 3270 or LU1 devices. For more information about the storage protection facilities available in CICS, see the *CICS System Definition Guide*.

3 The MODE statement specifies whether the map set is to be loaded above (RMODE(ANY)) or below (RMODE(24)) the 16MB line. RMODE(ANY) indicates that CICS can load the map set anywhere in virtual storage, but tries to load it above the 16MB line, if possible.

4 Use the NAME statement to specify the name of the physical map set that BMS loads into storage. If the map set is device-dependent, derive the map set name by appending the device suffix to the original 1- to 7-character map set name used in the application program. The suffixes to be appended for the various terminals supported by CICS BMS depend on the parameter specified in the TERM or SUFFIX operand of the DFHMSD macros used to define the map set. For programming information giving a complete list of map set suffixes, see the *CICS Application Programming Reference* manual.

To use a physical map set, you must define and install a resource definition for it. You can do this either by using the program autoinstall function or by using the CEDA DEFINE MAPSET and INSTALL commands, as described in “Defining programs, map sets, and partition sets to CICS” on page 134.

Installing symbolic description map sets

Symbolic description map sets enable the application programmer to make symbolic references to fields in the physical map set. Figure 29 shows the preparation of symbolic description map sets for BMS.

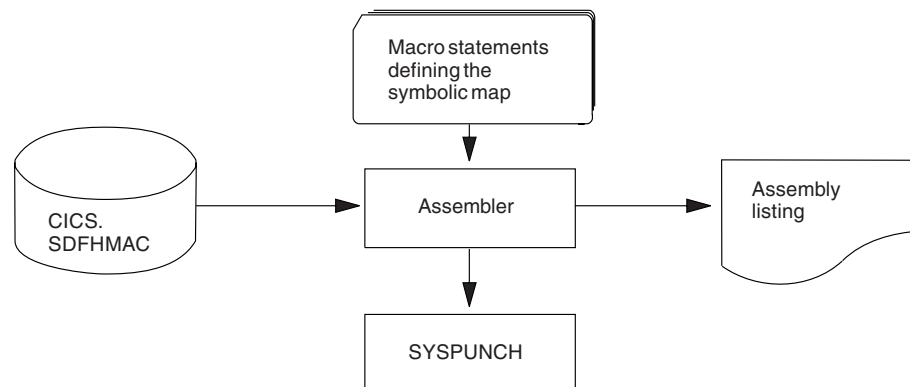


Figure 29. Installing symbolic description map sets using the DFHASMVS procedure

To use a symbolic description map set in a program, you must assemble the source statements for the map set and obtain a punched copy of the storage definition through SYSPUNCH. The first time this is done, you can direct the SYSPUNCH output to SYSOUT=A to get a listing of the symbolic description map set. If many map sets are to be used at your installation, or there are multiple users of common map sets, establish a private user copy library for each language that you use.

When a symbolic description is prepared under the same name for more than one programming language, a separate copy of the symbolic description map set must be placed in each user copy library. You must ensure that the user copy libraries are correctly concatenated with SYSLIB.

You need only one symbolic description map set corresponding to all the different suffixed versions of the physical map set. For example, to run the same application on terminals with different screen sizes, you would:

1. Define two map sets each with the same fields, but positioned to suit the screen sizes. Each map set has the same name but a different suffix, which would match the suffix specified for the terminal.
2. Assemble and link-edit the different physical map sets separately, but create only one symbolic description map set, because the symbolic description map set would be the same for all physical map sets.

You can use the sample job stream in Figure 30 to obtain a listing of a symbolic description map set. It applies to all the programming languages supported by CICS.

```
//DSECT    JOB  'accounting information',CLASS=A,MSGLEVEL=1
//ASM      EXEC PROC=DFHASMVS,PARM.ASSEM='SYSPARM(DSECT) '
//SYSPUNCH DD  SYSOUT=A
//SYSIN    DD  *
:
:      Macro statements defining the map set
:
/*
//
```

Figure 30. Listing of a symbolic description map set

If you want to assemble symbolic description map sets in which length fields are halfword-aligned, change the EXEC statement of the sample job in Figure 30 to the following:

```
//ASSEM EXEC PROC=DFHASMVS,PARM.ASSEM='SYSPARM(ADSECT) '
```

To obtain a punched copy of a symbolic description map set, code the //SYSPUNCH statement in the above example to direct output to the punch data stream. For example:

```
//SYSPUNCH DD  SYSOUT=B
```

To store a symbolic description map set in a private copy library, use job control statements similar to the following:

```
//SYSPUNCH DD  DSN=USER.MAPLIB.ASM(map set name),DISP=OLD
//SYSPUNCH DD  DSN=USER.MAPLIB.COB(map set name),DISP=OLD
//SYSPUNCH DD  DSN=USER.MAPLIB.PLI(map set name),DISP=OLD
```

Installing physical and symbolic description maps together

Figure 31 shows the DFHMAPS procedure for installing physical and symbolic description maps together. The DFHMAPS procedure consists of the following four steps, shown in Figure 31:

1. The BMS macros that you coded for the map set are added to a temporary sequential data set.
2. The macros are assembled to create the physical map set. The MAP option is coded in the SYSPARM global variable in the EXEC statement (PARM='SYSPARM(MAP)').
3. The physical map set is link-edited to the CICS load library.
4. Finally, the macros are assembled again, this time to produce the symbolic description map set. In this step, DSECT is coded in the SYSPARM global variable in the EXEC statement (PARM='SYSPARM(DSECT)'). Output is directed to the destination specified in the //SYSPUNCH DD statement. In the DFHMAPS procedure, that destination is the CICSTS31.CICS.SDFHMAC library.

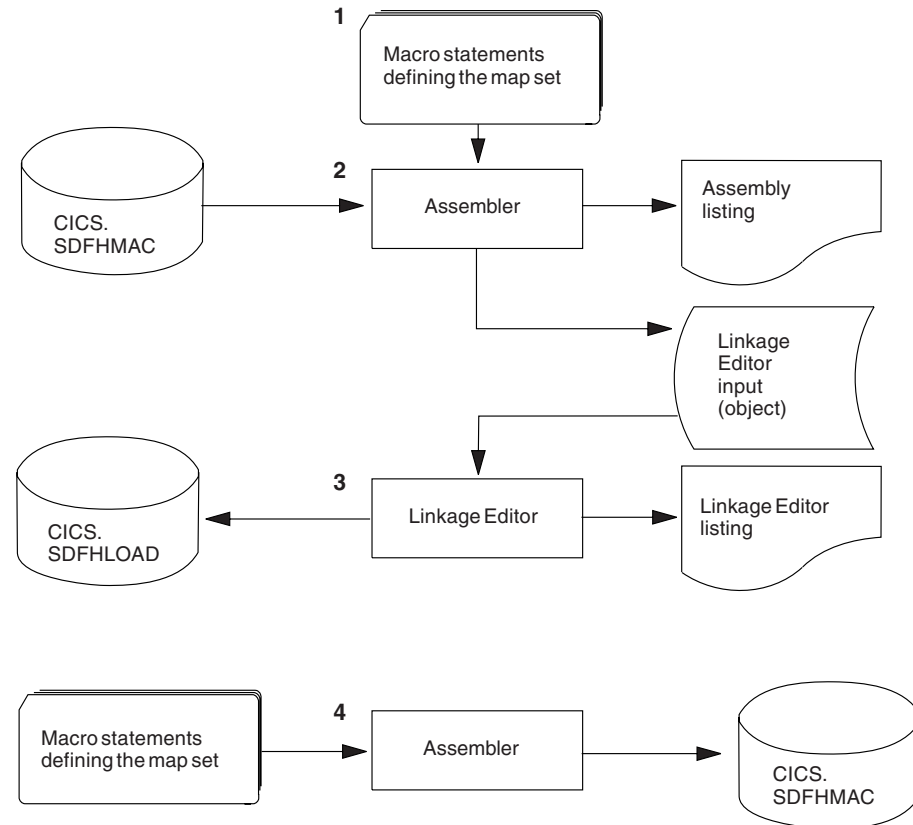


Figure 31. Installing a physical map set and a symbolic description map set together

Using the DFHMAPT procedure to install HTML templates from BMS maps

The DFHMAPT procedure is similar to DFHMAPS, with an additional step that installs HTML templates generated from the BMS maps. In this step, TEMPLATE is coded in the SYSPARM global variable in the EXEC statement (PARM='SYSPARM(TEMPLATE)'). In the DFHMAPT procedure, the output is directed to CICSTS31.CICS.SDFHHTML.

If you wish to use your own macro to customize HTML templates, and you do not wish to add your macro to the BMS source you should modify step ASMTEMPL:

1. Change the PARM parameter of the EXEC statement to
PARM='SYSPARM(TEMPLATE,macro_name),DECK,NOOBJECT'
2. Add the library that contains your macro to the SYSLIB concatenation.

JCL to install physical and symbolic description maps

The load module from the assembly of the physical map set and the source statements for the symbolic description map set can be produced in the same job by using the sample job stream in Figure 32.

```
//PREPARE JOB 'accounting information',CLASS=A,MSGLEVEL=1
//ASSEM EXEC PROC=DFHMAPS,MAPNAME=mapsetname,RMODE=ANY|24 (see note)
//SYSUT1 DD *
:
: Macro statements defining the map set
:
/*
//
```

Figure 32. Installing physical and symbolic description maps together

Note: The RMODE statement specifies whether the map set is to be loaded above (RMODE=ANY) or below (RMODE=24) the 16MB line. RMODE=ANY indicates that CICS can load the map set anywhere in virtual storage, but tries to load it above the 16MB line, if possible.

The DFHMAPS procedure produces map sets that are not halfword-aligned. If you want the length fields in input maps to be halfword-aligned, you have to code A=A on the EXEC statement. In the sample job in Figure 32, change the EXEC statement to:

```
//ASSEM EXEC PROC=DFHMAPS,MAPNAME=mapsetname,A=A
```

This change results in the SYSPARM operands in the assembly steps being altered to SYSPARM(AMAP) and SYSPARM(ADSECT) respectively.

The DFHMAPS procedure directs the symbolic description map set output (SYSPUNCH) to the CICSTS31.CICS.SDFHMAC library. Override this by specifying DSCTLIB=name on the EXEC statement, where “name” is the name of the chosen user copy library.

Adding a CSECT to your map assembly

It is possible that you might need to generate your BMS maps with a CSECT. For example, you might need to specify AMODE and RMODE options to ensure your maps reside above 16MB, or you might need to use the DFSMS binder IDENTIFY statement for reasons of change management. In this case, you need not only include the appropriate CSECT at the front of your BMS macro statements, but also add some conditional assembler statements to ensure that the CSECT statement is not included in the symbolic description map. The following example shows how you can add both a CSECT name and AMODE and RMODE statements:

```

//PREPARE JOB 'accounting information',CLASS=A,MSGLEVEL=1
//ASSEM EXEC PROC=DFHMAPS,MAPNAME=mapsetname,RMODE=ANY|24
//SYSUT1 DD *
.
AIF ('&SYSPARM' EQ 'DSECT').SKIPSD
AIF ('&SYSPARM' EQ 'ADSECT').SKIPSD
ANYNAME CSECT Binder IDENTIFY requires CSECT name
ANYNAME AMODE 31
ANYNAME RMODE ANY
.SKIPSD ANOP ,
DFH0STM DFHMSD TYPE=DSECT,MODE=INOUT,CTRL=FREEKB,LANG=COBOL, C
TIOAPFX=YES,TERM=3270-2,MAPATTS=(COLOR,HILIGHT), C
DSATTS=(COLOR,HILIGHT)
SPACE
DFH0STM DFHMDI SIZE=(24,80)
.
.
.
SPACE
DFHMSD TYPE=FINAL
END
.
/*
//

```

Figure 33. Adding a CSECT to the map assembly

Installing partition sets

Partition sets are installed in the same way as physical map sets (as illustrated in Figure 27 on page 128). There is no concept of a symbolic description partition set. The job stream in Figure 34 is an example of the assembly and link-edit of partition sets.

```

//PREP JOB 'accounting information',CLASS=A,MSGLEVEL=1
//STEP1 EXEC PROC=DFHASMVS
//SYSPUNCH DD DSN=&&TEMP,DCB=(RECFM=FB,BLKSIZE=2960),
// SPACE=(2960,(10,10)),UNIT=SYSDA,DISP=(NEW,PASS)
//SYSIN DD *
.
Macro statements defining the partition set
.
/*
//STEP2 EXEC PROC=DFHLNKVS,PARM='LIST,LET,XREF' 1
//SYSLIN DD DSN=&&TEMP,DISP=(OLD,DELETE)
// DD *
MODE RMODE(ANY|24) 2
NAME partitionsetname(R) 3
/*
//

```

Figure 34. Assembling and link-editing a partition set

Notes:

1 A partition set is loaded into CICS-key storage, unless it is link-edited with the RMODE(ANY) and RENT options. If it is link-edited with these options, it is loaded into key-0 protected storage, provided that RENTPGM=PROTECT is specified on the RENTPGM initialization parameter.

For more information about the storage protection facilities available in CICS, see the *CICS System Definition Guide*.

2 The MODE statement specifies whether the partition set is to be loaded above (RMODE(ANY)) or below (RMODE(24)) the 16MB line. RMODE(ANY) indicates that CICS can load the partition set anywhere in virtual storage, but tries to load it above the 16MB line, if possible.

3 Use the NAME statement to specify the name of the partition set which BMS loads into storage. If the partition set is device-dependent, derive the partition set name by appending the device suffix to the original 1- to 7-character partition set name used in the application program. The suffixes that BMS appends for the various terminals depend on the parameter specified in the SUFFIX operand of the DFHPSD macro that defined the partition set.

For programming information giving a complete list of partition-set suffixes, see the *CICS Application Programming Guide*.

To use a partition set, you must define and install a resource definition for it. You can do this either by using the program autoinstall function or by using the CEDA DEFINE PARTITIONSET and INSTALL commands, as described in the *CICS Resource Definition Guide*.

Defining programs, map sets, and partition sets to CICS

To be able to use a program that you have installed in one of the load libraries specified in your CICS startup JCL, the program, and any map sets and partition sets that it uses, must be defined to CICS. To do this, CICS uses the resource definitions MAPSET (for map sets), PARTITIONSET (for partition sets), and PROGRAM (for programs). You can create and install such resource definitions in any of the following ways:

- CICS can dynamically create, install, and catalog a definition for the program, map set, or partition set when it is first loaded, by using the autoinstall for programs function.
- You can create a specific resource definition for the program, map set, or partition set and install that resource definition in your CICS region.

You can install resource definitions in either of the following ways:

- At CICS initialization, by including the resource definition group in the group list specified on the GRPLIST system initialization parameter.
- While CICS is running, by the CEDA INSTALL command.

For information about defining programs to CICS, see the *CICS Resource Definition Guide*.

Part 3. Application design

Chapter 11. Application design	139
Pseudoconversational and conversational design	140
Terminal interruptibility	142
How tasks are started	142
Which transaction?	143
Separating business and presentation logic	146
Multithreading: Reentrant, quasi-reentrant and threadsafe programs	147
Quasi-reentrant application programs	147
Threadsafe programs	149
Threadsafe considerations for statically or dynamically called routines	153
OPENAPI programs	153
Obligations of OPENAPI programs	154
Using the FORCEQR system initialization parameter	155
Non-reentrant programs	155
Storing data within a transaction	156
Transaction work area (TWA)	156
User storage	156
COMMAREA in LINK and XCTL commands	157
Channels in LINK and XCTL commands	158
Program storage	158
Temporary storage	158
Intrapartition transient data	160
GETMAIN SHARED command	160
Your own data sets	160
Lengths of areas passed to CICS commands	162
LENGTH options	162
Journal records	162
Data set definitions	162
Recommendation	162
Minimizing errors	163
Protecting CICS from application errors	163
Testing applications	163
Non-terminal transaction security	164
 Chapter 12. Design for performance	 165
Program size	165
Virtual storage	166
Reducing paging effects	167
Locality of reference	167
Working set	168
Reference set	168
Exclusive control of resources	169
Operational control	170
Operating system waits	171
The NOSUSPEND option	171
Efficient data operations	172
Efficient database operations	172
Efficient data set operations	172
VSAM data sets	173
BDAM data sets	174
Efficient browsing (in non-RLS mode)	174
Efficient logging	175
Efficient sequential data set access	175

Efficient terminal operations.	176
Length of the data stream sent to the terminal.	176
Basic mapping support considerations.	176
Avoid turning on modified data tags (MDTs) unnecessarily.	176
Use FRSET to reduce inbound traffic.	177
Do not send blank fields to the screen.	177
Address CICS areas correctly.	177
Use the MAPONLY option when possible.	177
Send only changed fields to an existing screen.	177
Design data entry operations to reduce line traffic.	178
Compress data sent to the screen.	178
Use nulls instead of blanks.	178
Use methods that avoid the need for nulls or blanks.	179
Page-building and routing operations.	179
Sending multipage output.	179
Sending messages to destinations other than the input terminal.	180
Sending pages built from multiple maps.	180
Using the BMS page-copy facility.	180
Requests for printed output.	181
Additional terminal control considerations.	181
Use only one physical SEND command per screen.	181
Use the CONVERSE command.	181
Limit the use of message integrity options.	181
Avoid using the DEFRESP option on SEND commands.	182
Avoid using unnecessary transactions.	182
Send unformatted data without maps.	182
 Chapter 13. Sharing data across transactions.	 183
Using the common work area (CWA).	183
Protecting the CWA.	184
Using the TCTTE user area (TCTUA).	186
Using the COMMAREA in RETURN commands.	187
Using a channel on RETURN commands.	187
Using the display screen to share data.	188
 Chapter 14. Enhanced inter-program data transfer: channels as modern-day COMMAREAs.	 189
Channels: quick start.	189
Containers and channels.	189
Basic examples.	190
Using channels: some typical scenarios.	192
One channel, one program.	192
One channel, several programs (a component).	193
Several channels, one component.	193
Multiple interactive components.	194
Creating a channel.	195
The current channel.	196
Current channel example, with LINK commands.	196
Current channel example, with XCTL commands.	198
Current channel: START and RETURN commands.	199
The scope of a channel.	200
Scope example, with LINK commands.	200
Scope example, with LINK and XCTL commands.	202
Discovering which containers a program's been passed.	204
Discovering which containers were returned from a link.	204
CICS read only containers.	204

#

Designing a channel: best practices	205
Constructing and using a channel: an example.	206
Channels and BTS activities	207
Context	208
Using channels from JCICS.	209
Dynamic routing with channels	209
Data conversion	210
Why is data conversion needed?	210
Preparing for code page conversion with channels	210
Data conversion with channels	212
How to cause CICS to convert character data automatically	213
Using containers to do code page conversion	214
A SOAP example	214
Benefits of channels	215
Migrating from COMMAREAs to channels	216
Migration of existing functions	216
Migration to the new function	216
Migrating LINK commands that pass COMMAREAs	217
Migrating XCTL commands that pass COMMAREAs	217
Migrating pseudoconversational COMMAREAs on RETURN commands	218
Migrating START data	218
Migrating programs that use temporary storage to pass data	218
Migrating dynamically-routed applications.	219
Chapter 15. Affinity	221
Types of affinity	222
Inter-transaction affinity	222
Transaction-system affinity	222
Using INQUIRE and SET commands and global user exits	222
Programming techniques and affinity	223
Safe techniques	223
Unsafe techniques	223
Suspect techniques.	224
Recommendations	224
Safe programming to avoid affinity	224
The COMMAREA	225
The TCTUA	226
Using the TCTUA in an unsafe way	227
Using ENQ and DEQ commands with ENQMODEL resource definitions	228
Overview of sysplex enqueue and dequeue	228
Benefits	229
BTS containers	229
Unsafe programming for affinity	229
Using the common work area	229
Using GETMAIN SHARED storage	230
Using the LOAD PROGRAM HOLD command	231
Sharing task-lifetime storage	232
Using the WAIT EVENT command	234
Using ENQ and DEQ commands without ENQMODEL resource definitions	235
Suspect programming for affinity	236
Using temporary storage	236
Naming conventions for remote queues	237
Exception conditions for globally accessible queues	238
Using transient data	239
Exception conditions for globally accessible queues	239
Using the RETRIEVE WAIT and START commands	240

Using the START and CANCEL REQID commands	241
Using the DELAY and CANCEL REQID commands	243
Using the POST and CANCEL REQID commands	244
Detecting inter-transaction affinities	246
Inter-transaction affinities caused by application generators	246
Duration and scope of inter-transaction affinities	246
Affinity transaction groups	247
Relations and lifetimes	247
The global relation	247
The LUname (terminal) relation	248
The userid relation	250
The BAPPL relation.	251
Chapter 16. Recovery design	255
Journaling	255
Journal records	255
Journal output synchronization.	255
Syncpointing	257
Chapter 17. Dealing with exception conditions	261
Default CICS exception handling	261
Handling exception conditions by in-line code	262
How to use the RESP and RESP2 options	262
Use of RESP and DFHRESP in COBOL and PL/I.	262
Use of RESP and DFHRESP in C and C++.	263
Use of DFHRESP in assembler	263
An example of exception handling in C	263
An example of exception handling in COBOL	264
Modifying default CICS exception handling	265
Using the HANDLE CONDITION command	267
RESP and NOHANDLE options	268
How CICS keeps track of what to do	268
Using the HANDLE CONDITION ERROR command.	269
Using the IGNORE CONDITION command	269
Using the HANDLE ABEND command.	270
Using PUSH HANDLE and POP HANDLE commands	271
Chapter 18. Abnormal termination recovery	273
Creating a program-level abend exit	274
Retrying operations.	275
Trace	276
Trace entry points	277
System trace entry points	277
User trace entry points	277
Exception trace entry points.	277
User exception trace entry points.	277
Monitoring	277
Dump	278
Chapter 19. The QUERY SECURITY command	281
Using the QUERY SECURITY command	281
Security protection at the record or field level	281
CICS-defined resource identifiers.	282
SEC system initialization parameter.	282
Programming hints	282

Chapter 11. Application design

This chapter introduces some basic concepts to help you design CICS applications. Changes are suggested that can improve performance and efficiency, but further guidance on programming for efficiency is provided in Chapter 12, “Design for performance,” on page 165.

The programming models implemented in CICS are inherited from those designed for 3270s, and exhibit many of the characteristics of conversational, terminal-oriented applications. There are basically three styles of programming model:

- Terminal-initiated, that is, the conversational model
- Distributed program link (DPL), or the RPC model
- START, that is, the queuing model.

Once initiated, the applications typically use these and other methods of continuing and distributing themselves, for example, with pseudoconversations, RETURN IMMEDIATE or DTP. The main difference between these models is in the way that they maintain state (for example, security), and hence state becomes an integral part of the application design. This presents the biggest problem when you attempt to convert to another application model.

A pseudoconversational model is mostly associated with terminal-initiated transactions and was developed as an efficient implementation of the conversational model. With increased use of 1-in and 1-out protocols such as HTTP, it is becoming necessary to add the pseudoconversational characteristic to the DPL or RPC model.

- “Pseudoconversational and conversational design” on page 140
- “How tasks are started” on page 142
- “Which transaction?” on page 143
- “Separating business and presentation logic” on page 146
- “Multithreading: Reentrant, quasi-reentrant and threadsafe programs” on page 147
- “Storing data within a transaction” on page 156
- “Lengths of areas passed to CICS commands” on page 162
- “Minimizing errors” on page 163
- “Non-terminal transaction security” on page 164

Pseudoconversational and conversational design

In a **conversational** transaction, the length of time spent in processing each of a user's responses is extremely short when compared to the amount of time waiting for the input. A conversational transaction is one that involves more than one input from the terminal, so that the transaction and the user enter into a conversation. A **nonconversational** transaction has only one input (the one that causes the transaction to be invoked). It processes that input, responds to the terminal and terminates.

Processor speeds, even allowing for accessing data sets, are considerably faster than terminal transmission times, which are considerably faster than user response times. This is especially true if users have to think about the entry or have to enter many characters of input. Consequently, conversational transactions tie up storage and other resources for much longer than nonconversational transactions.

A **pseudoconversational** transaction sequence contains a series of nonconversational transactions that look to the user like a single conversational transaction involving several screens of input. Each transaction in the sequence handles one input, sends back the response, and terminates.

Before a pseudoconversational transaction terminates, it can pass data forward to be used by the next transaction initiated from the same terminal, whenever that transaction arrives. A pseudoconversational transaction can specify what the next transaction is to be, using the TRANSID option of the RETURN command. However, you should be aware that if another transaction is started for that device, it may interrupt the pseudoconversational chain you have designed, unless you specify the IMMEDIATE option on the RETURN command. In this case, the transaction specified by the TRANSID command is attached regardless of any other transactions queued for this terminal.

The RETURN command is described in the *CICS Application Programming Reference* manual.

No transaction exists for the terminal from the time a response is written until the user sends the next input and CICS starts the next transaction to respond to it. Information that would normally be stored in the program between inputs is passed from one transaction in the sequence to the next using the COMMAREA or one of the other facilities that CICS provides for this purpose. (See Chapter 13, "Sharing data across transactions," on page 183 for details.)

There are two major issues to consider in choosing between conversational and pseudoconversational programming.

- The effect of the transaction on **contention** resources, such as storage and processor usage. Storage is required for control blocks, data areas, and programs that make up a transaction, and the processor is required to start, process, and terminate tasks. Conversational programs have a **very** high impact on storage, because they last so long, relative to the sum of the transactions that make up an equivalent pseudoconversational sequence. However, there is less processor overhead, because only one transaction is initiated instead of one for every input.
- The effect on **exclusive-use** resources, such as records in recoverable data sets, recoverable transient data queues, enqueue items, and so on. Again, a conversational transaction holds on to these resources for much longer than the corresponding sequence of nonconversational transactions. From this point of

view, pseudoconversational transactions are better for quick responses, but recovery and integrity implications may mean that you prefer to use conversational transactions.

To summarize, although conversational tasks may be easier to write, they have serious disadvantages—both in performance (especially the need for virtual storage) and in their effect on the overall operability of the CICS systems containing them. Processors are now larger, with more real storage and more power than in the past, and this makes conversational tasks less painful in small amounts; but if you use conversational applications, you may rapidly run into virtual storage constraint. If you run application programs above the line, you will probably encounter ENQ problems before running into virtual storage constraints.

CICS ensures that changes to recoverable resources (such as data sets, transient data, and temporary storage) made by a unit of work (UOW) are made completely or not at all. A UOW is equivalent to a transaction, unless that transaction issues SYNCPOINT commands, in which case a UOW lasts between syncpoints. For a more detailed description of syncpoints and UOWs, see the *CICS Recovery and Restart Guide*.

When a transaction makes a change to a recoverable resource, CICS makes that resource unavailable to any other transaction that wants to change it until the original transaction has completed. In the case of a conversational transaction, the resources in question may be unavailable to other terminals for relatively long periods.

For example, if one user tries to update a particular record in a recoverable data set, and another user tries to do so before the first one finishes, the second user's transaction is suspended. This has advantages and disadvantages. You would not want the second user to begin updating the record while the first user is changing it, because one of them is working from what is about to become an obsolete version of the record, and these changes erase the other user's changes. On the other hand, you also do not want the second user to experience the long, unexplained wait that occurs when that transaction attempts to READ for UPDATE the record that is being changed.

If you use pseudoconversational transactions, however, the resources are only very briefly unavailable (that is, during the short component transactions). However, unless all recoverable resources can be updated in just one of these transactions, recovery is impossible because UOWs cannot extend across transactions. So, if you cannot isolate updates to recoverable resources in this way, you must use conversational transactions.

The previous example poses a further problem for pseudoconversational transactions. Although you could confine all updating to the final transaction of the sequence, there is nothing to prevent a second user from beginning an update transaction against the same record while the first user is still entering changes. This means that you need additional application logic to ensure integrity. You can use some form of enqueueing, or you can have the transaction compare the original version of the record with the current version before actually applying the update.

Terminal interruptibility

When a conversational task is running, CICS allows nothing else to send messages to that task's terminal. This has advantages and disadvantages. The advantage is that unexpected messages (for example, broadcasts) cannot interrupt the user-machine dialogue and, worse, corrupt the formatted screen. The disadvantage is that the end user cannot then be informed of important information, such as the intention of the control operator to shut down CICS after 10 minutes. More importantly, the unwitting failure of the end user to terminate the conversation may in fact prevent or delay a normal CICS shutdown.

Pseudoconversational applications can allow messages to come through between message pairs of a conversation. This means that notices like shutdown warnings can be delivered. This might disturb the display screen contents, and can sometimes interfere with transaction sequences controlled by the RETURN command with the TRANSID option. However, this can be prevented by using the IMMEDIATE option, or by forcing the terminal into NOATI status during the middle of a linked sequence of interactions.

How tasks are started

Work is started in CICS—that is, tasks are initiated—in one of two ways:

1. From unsolicited input
2. By automatic task initiation (ATI)

Automatic task initiation occurs when:

- An existing task asks CICS to create another one. The START command, the IMMEDIATE option on a RETURN command (discussed in “RETURN IMMEDIATE” on page 367), and the SEND PAGE command (in “The SEND PAGE command” on page 508) all do this.
- CICS creates a task to process a transient data queue (see “Automatic transaction initiation (ATI)” on page 595).
- CICS creates a task to deliver a message sent by a BMS ROUTE request (see Chapter 39, “Message routing,” on page 527). The CSPG tasks you see after using the CICS-supplied transaction CMSG are an example of this. CMSG uses a ROUTE command which creates a CSPG transaction for each target terminal in your destination list.

The primary mechanism for initiating tasks, however, is unsolicited input. When a user transmits input from a terminal which is not the **principal facility** of an existing task, CICS creates a task to process it. The terminal that sent the input becomes the principal facility of the new task.

Principal facility

CICS allows a task to communicate directly with only one terminal, namely its principal facility. CICS assigns the principal facility when it initiates the task, and the task “owns” the facility for its duration. No other task can use that terminal until the owning task ends. If a task needs to communicate with a terminal other than its principal facility, it must do so indirectly, by creating another task that has the terminal as its principal facility. This requirement arises most commonly in connection with printing, and how you can create such a task is explained in “Using CICS printers” on page 415.

Notes:

1. You can specify a terminal destination other than your principal facility in a SEND command if the destination is under TCAM control, an apparent

exception to this rule. This is possible because communications with TCAM terminals are always queued. Thus your task does not write directly to the destination terminal, but instead writes to a queue that will be delivered to it subsequently by TCAM (see “Using TCAM” on page 383) . BMS routing, described in Chapter 39, “Message routing,” on page 527, is another form of indirect access to other terminals by queues.

2. In CICS TS 3.1, local TCAM terminals are not supported. The only TCAM terminals supported are remote terminals connected to a pre-CICS TS 3.1 terminal-owning region by the DCB (not ACB) interface of TCAM.

Unsolicited inputs from other systems are handled in the same way: CICS creates a task to process the input, and assigns the conversation over which the input arrived as the principal facility. (Thus a conversation with another system may be either a principal or alternate facility. In the case where a task in one CICS region initiates a conversation with another CICS region, the conversation is an alternate facility of the initiating task, but the principal facility of the partner task created by the receiving system. By contrast, a terminal is always the principal facility.)

Alternate facility

Although a task may communicate directly with only one terminal, it can also establish communications with one or more remote systems. It does this by asking CICS to assign a conversation with that system to it as an **alternate facility**. The task “owns” its alternate facilities in the same way that it owns its principal facility. Ownership lasts from the point of assignment until task end or until the task releases the facility.

Not all tasks have a principal facility. Tasks that result from unsolicited input always do, by definition, but a task that comes about from automatic task initiation may or may not need one. When it does, CICS waits to initiate the task until the requested facility is available for assignment to the task.

Which transaction?

Having received an unsolicited input, how does CICS decide what to do with it? That is, what transaction should the task created to process it execute? The short answer is that the previous task with the same principal facility usually tells CICS what transaction to execute next just before it ends, by the TRANSID option on its final RETURN. This is almost always the case in a pseudoconversational transaction sequence, and usually in menu-driven applications as well. Failing that, and in any case to get a sequence started, CICS interprets the first few characters of the input as a transaction code. However, it is more complicated than that; the exact process goes as follows. The step numbers indicate the order in which the tests are made and refer to Figure 35 on page 144, a diagram of this logic.

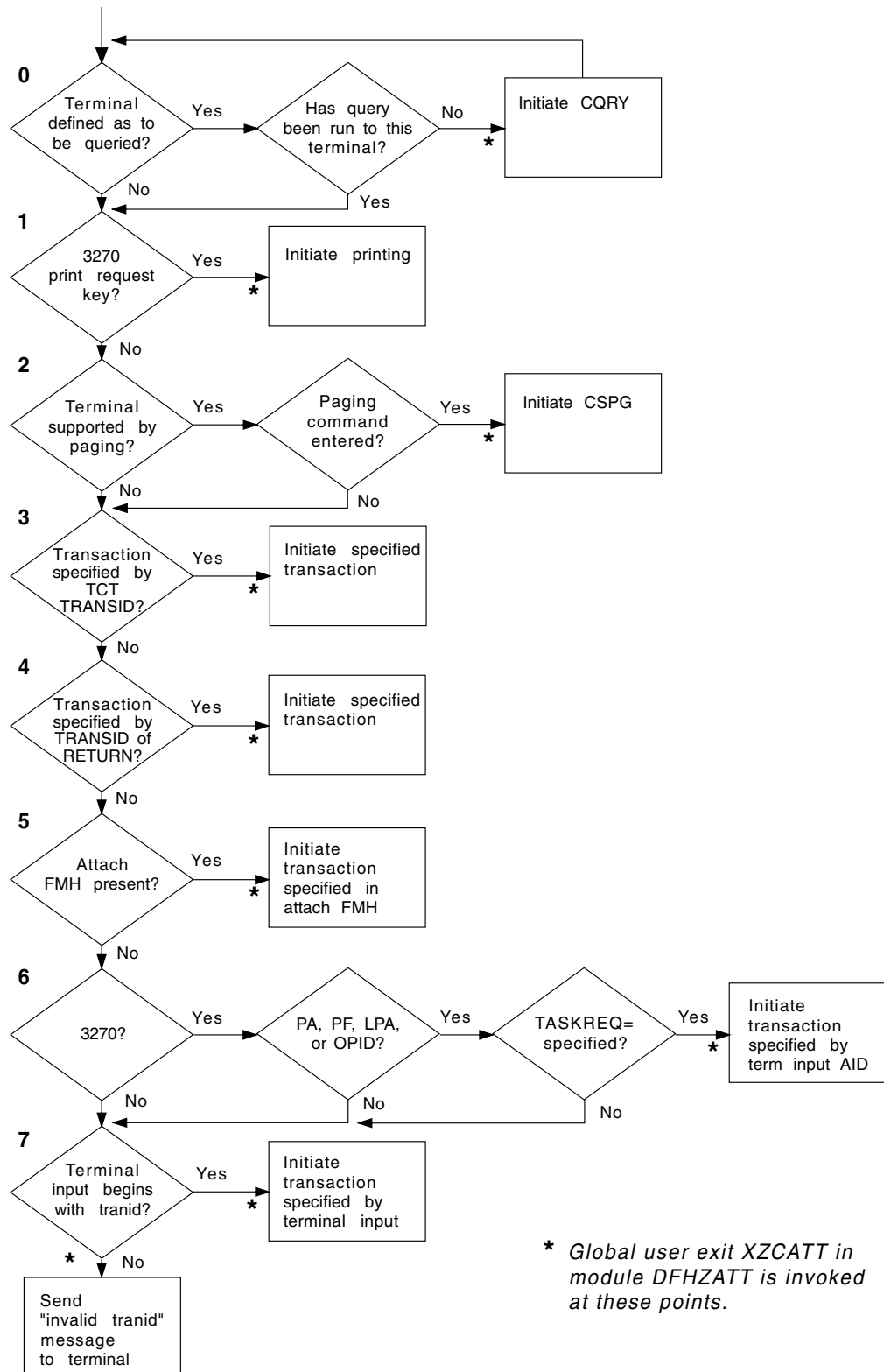


Figure 35. Determining which transaction to execute

0. On the very first input from a terminal, CICS sometimes schedules a preliminary task before creating one to process the input. This task executes the CICS-supplied “query” transaction, CQRY, which causes the

terminal to transmit an encoded description of some of its hardware characteristics—extended attributes, character sets, and so on.

CQRY allows the system programmer to simplify maintenance of the terminal network by omitting these particulars from the terminal definitions. It occurs only if the terminal definition so specifies, and has no effect on the subsequent determination of what transaction to use to process the input, which goes as follows.

1. If the terminal is a 3270 and the input is the “print request key”, the CICS-supplied transaction that prints the contents of the screen, CSPP, is initiated. See “CICS print key” on page 420 for more information about this feature. For this purpose, a “3270 logical unit” or any other device that accepts the 3270 data stream counts as a 3270.
2. If full BMS support is present, the terminal is of a type supported by BMS terminal paging, and the input is a paging command, the CICS-supplied transaction CSPG is initiated to process the request. BMS support levels are explained in “BMS support levels” on page 453, and the same section contains a list of the terminals that BMS supports. The PGRET, SKRxxxx, PGCHAIN, PGCOPY, and PGPURGE options in the system initialization table define the paging commands. As paging requires full BMS, this step is skipped if the CICS system contains less than that level.
3. If the terminal definition indicates that a specific transaction should be used to process all unsolicited inputs from that terminal, the indicated transaction is executed. (If present, this information appears in the TRANSACTION attribute of the TERMINAL definition.)
4. If the previous task at the terminal specified the TRANSID option of the RETURN command that ended it, the transaction named is executed.
5. If an attach function management header is present in the input, the attach names in the header are converted to a 4-character CICS transaction identifier, and that transaction is executed.
6. If the terminal is a 3270, and the attention identifier is defined as a transaction, that transaction is executed. “Attention keys” on page 401 explains attention identifiers. You define one as a transaction identifier with the TASKREQ attribute of the corresponding TRANSACTION definition.
7. If all of the preceding tests fail, the initial characters of the input are used to identify the transaction to be executed. The characters used are the first ones (up to four) after any control information in the data stream and before the first field separator character or the next 3270 Control Character (X'00' to X'3F'). Field separators are defined in the FLDSEP option of the system initialization table (the default is a blank).

If there are no such characters in the input, as occurs when you use the CLEAR key, for example, or if there is no transaction definition that matches the input, CICS cannot determine what transaction to execute and sends an “invalid transaction identification” message to the terminal.

Note: This logic for deciding which transaction to execute applies only to tasks initiated to process unsolicited inputs. For automatic transaction initiation, the transaction is always known. You specify it in the TRANSID option when you create a task with a START or RETURN IMMEDIATE. Similarly, you specify what transaction should be used to process a transient data queue in the queue definition. Tasks created to route messages always execute the CICS-supplied transaction CSPG.

Separating business and presentation logic

In general, it is good practice to split applications into a part containing the business code that is reusable, and a part responsible for presentation to the client. This technique enables you to improve performance by optimizing the parts separately, and allows you to reuse your business logic with different forms of presentation.

When separating the business and presentation logic, you need to consider the following:

- Avoid affinities between the two parts of the application.
- Be aware of the DPL-restricted API; see *CICS Application Programming Reference* for details.
- Be aware of hidden presentation dependencies, such as EIBTRMID usage.

Figure 36 illustrates a simple CICS application that accepts data from an end user, updates a record in a file, and sends a response back to the end user. The transaction that runs this program is the second in a pseudoconversation. The first transaction has sent a BMS map to the end user's terminal, and the second transaction reads the data with the EXEC CICS RECEIVE MAP command, updates the record in the file, and sends the response with the EXEC CICS SEND MAP command.

The EXEC CICS RECEIVE and EXEC CICS SEND MAP commands are part of the transaction's presentation logic, while the EXEC CICS READ UPDATE and EXEC CICS REWRITE commands are part of the business logic.

```
..  
EXEC CICS RECEIVE MAP  
..  
EXEC CICS READ UPDATE  
..  
EXEC CICS REWRITE  
..  
EXEC CICS SEND MAP  
..
```

Figure 36. CICS functions in a single application program

A sound principle of modular programming in CICS application design is to separate the presentation logic from the business logic, and to use a communication area and the EXEC CICS LINK command to make them into a single transaction. Figure 37 and Figure 38 on page 147 illustrate this approach to application design.

```
..  
EXEC CICS RECEIVE MAP  
..  
EXEC CICS LINK..  
..  
EXEC CICS SEND MAP  
..
```

Figure 37. Presentation logic

```

..
EXEC CICS ADDRESS COMMAREA
..
EXEC CICS READ UPDATE
..
EXEC CICS REWRITE
..
EXEC CICS RETURN..

```

Figure 38. Business logic

Once the business logic of a transaction has been isolated from the presentation logic and given a communication area interface, it is available for reuse with different presentation methods. For example, you could use Distributed Program Link (DPL) to implement a two-tier model, or CICS Web support with the CICS business logic interface, where the presentation logic is HTTP-based.

Multithreading: Reentrant, quasi-reentrant and threadsafe programs

Multithreading is a technique that allows a single copy of an application program to be processed by several transactions concurrently. For example, one transaction may begin to execute an application program. When an EXEC CICS command is reached, causing a CICS WAIT and call to the dispatcher, another transaction may then execute the same copy of the application program. (Compare this with single-threading, which is the execution of a program to completion: processing of the program by one transaction is completed before another transaction can use it.)

Multithreading requires that all CICS application programs be quasi-reentrant; that is, they must be serially reusable between entry and exit points. CICS application programs using the EXEC CICS interface obey this rule automatically. For COBOL, C, and C++ programs, reentrancy is ensured by a fresh copy of working storage being obtained each time the program is invoked. You should always use the RENT option on the compile or pre-link utility even for C and C++ programs that do not have writable statics and are naturally reentrant. Temporary variables and DFHEIPTR fields inserted by the CICS translator are usually defined as writable static variables and require the RENT option. For these programs to stay reentrant, variable data should not appear as static storage in PL/I, or as a DC in the program CSECT in assembler language.

As well as requiring that your application programs are compiled and link-edited as reentrant, CICS also identifies programs as being either quasi-reentrant or threadsafe. These attributes are discussed in the following sections.

Quasi-reentrant application programs

CICS runs user programs under a CICS-managed task control block (TCB). If your programs are defined as quasi-reentrant (on the CONCURRENCY attribute of the program resource definition), CICS always invokes them under the CICS quasi-reentrant (QR) TCB. The requirements for a quasi-reentrant program in a multithreading context are less stringent than if the program were to execute concurrently on multiple TCBs.

CICS requires that an application program is reentrant so that it guarantees consistent conditions. In practice, an application program may not be truly reentrant; CICS expects “quasi-reentrancy”. This means that the application program should be in a consistent state when control is passed to it, both on entry, and before and after each EXEC CICS command. Such quasi-reentrancy guarantees that each

invocation of an application program is unaffected by previous runs, or by concurrent multi-threading through the program by multiple CICS tasks.

For example, application programs could modify their executable code, or the variables defined within the program storage, but these changes must be undone, or the code and variables reinitialized, before there is any possibility of the task losing control and another task executing the same program.

CICS quasi-reentrant user programs (application programs, user-replaceable modules, global user exits, and task-related user exits) are given control by the CICS dispatcher under the QR TCB. When running under this TCB, a program can be sure that no other quasi-reentrant program can run until it relinquishes control during a CICS request, at which point the user task is suspended, leaving the program still “in use”. The same program can then be reinvoked for another task, which means the application program can be in use concurrently by more than one task, although only one task at a time can actually be executing.

To ensure that programs cannot interfere with each others working storage, CICS obtains a separate copy of working storage for each execution of an application program. Thus, if a user application program is in use by 11 user tasks, there are 11 copies of working storage in the appropriate dynamic storage area (DSA).

Quasi-reentrancy allows programs to access globally shared resources—for example, the CICS common work area (CWA)—without the need to protect those resources from concurrent access by other programs. Such resources are effectively locked exclusively to the running program, until it issues its next CICS request. Thus, for example, an application can update a field in the CWA without using compare and swap (CS) instructions or locking (enqueueing on) the resource.

Note: The CICS QR TCB provides protection through exclusive control of global resources only if all user tasks that access those resources run under the QR TCB. It does not provide automatic protection from other tasks that execute concurrently under another (open) TCB.

Take care if a program involves lengthy calculations: because an application program retains control from one EXEC CICS command to the next, the processing of other transactions on the QR TCB is completely excluded. However, you can use the task-control SUSPEND command to allow other transaction processing to proceed; see Chapter 44, “Task control,” on page 561 for details. Note that runaway task time interval is controlled by the transaction definition and the system initialization parameter ICVR. CICS purges a task that does not return control before expiry of the ICVR-specified interval.

Threadsafe programs

In the CICS open transaction environment (OTE), when application programs, task-related user exits (TRUEs), global user exit programs, and user-replaceable modules are defined to CICS as threadsafe, they can run concurrently on open TCBs. Because of this, they cannot rely on quasi-reentrancy to protect shared resources from concurrent access by another program. Furthermore, quasi-reentrant programs might also be placed at risk if they access shared resources that can also be accessed by a user task running concurrently under an open TCB. The techniques used by user programs to access shared resources must therefore take into account the possibility of simultaneous access by other programs. To gain the performance benefits of the open transaction environment while maintaining the integrity of shared resources, serialization techniques must be used to prohibit concurrent access to shared resources. Programs that use appropriate serialization techniques when accessing shared resources are described as threadsafe. (The term fully reentrant is also used sometimes, but this can be misunderstood, hence threadsafe is the preferred term.)

The performance benefits of being threadsafe can only be gained by applications that involve a task-related user exit (TRUE) enabled using the OPENAPI option on the ENABLE PROGRAM command. Task-related user exits like this are known as open API TRUEs. An open API TRUE will be given control under an L8 mode open TCB, and can use non-CICS APIs without having to create, manage and switch between subtask TCBs. The CICS DB2 task-related user exit that is used by the CICS DB2 attachment facility operates as an open API TRUE when CICS is connected to DB2 Version 6 or later, which means that CICS DB2 applications can gain performance benefits from being threadsafe (as explained in “SQL, threadsafe and other programming considerations for CICS DB2 applications” in the *CICS DB2 Guide*).

The goal of making programs threadsafe is to enable them to remain on an open TCB, rather than switching back and forth between the open TCB and the QR TCB. When CICS is connected to DB2 Version 6 or later, TCB switching occurs in the following circumstances:

- When a program that is not defined as threadsafe makes a DB2 request, CICS switches from the QR TCB (where the program is executing) to an open TCB, and back to the QR TCB again when the DB2 request is complete.
- When a user exit program that is not defined as threadsafe is used in the course of a DB2 request, CICS switches from the open TCB (where the DB2 request is executing) to the QR TCB. The user exit program is executed on the QR TCB, and then the task is switched back to the open TCB to complete the DB2 request. For example, the XRMIIN and XRMIOU global user exits might be invoked in the course of the DB2 request. If the exit programs are not defined as threadsafe, this TCB switching occurs. If the exit programs are defined as threadsafe, processing will continue throughout on the open TCB.
- When a program that is defined as threadsafe and is executing on an open TCB invokes any EXEC CICS commands which are not threadsafe, CICS switches back from the open TCB to the QR TCB to execute the non-threadsafe code. The program then continues to execute on the QR TCB. If the program does not make any further DB2 requests, then the switch back to the QR TCB is only a disadvantage because it increases the usage of your QR TCB for the time taken to run any remaining application code. However, if the program makes any further DB2 requests, CICS must switch back again to the open TCB.
- When a program that is defined as threadsafe and is executing on an open TCB invokes a task-related user exit program which is not defined as threadsafe,

CICS switches back to the QR TCB and gives control to the task-related user exit program. When the task-related user exit program completes processing, the application program continues to execute on the QR TCB, in the same way as it would after issuing a non-threadsafe EXEC CICS command.

- When a program that is defined as threadsafe and is executing on an open TCB invokes a threadsafe CICS command, it is possible for a global user exit to be invoked as part of executing the command. If a global user exit program is used which is not defined as threadsafe, CICS switches back to the QR TCB and gives control to the global user exit program. When the user exit program completes processing, CICS switches back to the open TCB to continue processing the threadsafe CICS command.
- When a program that is defined as threadsafe and is executing on an open TCB completes, CICS switches back to the QR TCB for task termination. This switch is always necessary.

The maximum TCB switching for a CICS DB2 application would occur if your program used a non-threadsafe user exit program and a non-threadsafe EXEC CICS command after every DB2 request. In particular, the use of a non-threadsafe exit program on the CICS-DB2 mainline path (for example, a program that is enabled at XRMIIN or XRMIOUT) causes more TCB switching than what is experienced when CICS is connected to Version 5 or earlier.

If you want to make an application program remain on an open TCB:

1. **Ensure that the program's logic is threadsafe.** That is, the native language code in between the EXEC CICS commands must be threadsafe. If you define a program to CICS as threadsafe but include application logic that is not threadsafe, the results are unpredictable, and CICS is not able to protect you from the possible consequences. Later in this topic, we have more information about producing threadsafe application logic.
2. **Ensure that the program uses only threadsafe EXEC CICS commands.** The commands that are threadsafe are indicated in the command syntax diagrams in the *CICS Application Programming Reference* and the *CICS System Programming Reference* with the statement "This command is threadsafe", and are listed in "Threadsafe command list" in the *CICS Application Programming Reference* and Appendix D of the *CICS System Programming Reference*. If you include a non-threadsafe EXEC CICS command in a program which is running on an open TCB, CICS switches back from the open TCB to the QR TCB to ensure that the command is processed safely. The TCB switching could be detrimental to the application's performance.

As well as checking EXEC CICS commands that you code explicitly, be aware
of high-level language constructs or Language Environment callable services
used by your program that result in using CICS services. CICS services used in
this way might involve non-threadsafe CICS commands, and cause a switch
back to the QR TCB. In particular, the COBOL statement DISPLAY UPON
SYSOUT, some types of PL/I and C++ output, and the Language Environment
callable services CEEMOUT and CEE3DMP, write data to the Language
Environment transient data destinations CESE and CESO. This involves an
EXEC CICS WRITE TD command, which is not threadsafe.

3. **Ensure that the program is defined to CICS as threadsafe.** Use the CONCURRENCY attribute of the program resource definition to do this. By defining a program to CICS as threadsafe, you are only specifying that the application logic is threadsafe, not that all the EXEC CICS commands included in the program are threadsafe. CICS can ensure that EXEC CICS commands

are processed safely by using TCB switching. In order to permit your program to run on an open TCB, CICS needs you to guarantee that your application logic is threadsafe.

4. **Ensure that any user exit programs in the execution path used by the program are coded to threadsafe standards and defined to CICS as threadsafe.** This might include dynamic plan exits, global user exits, or task-related user exits. (Note for task-related user exits, enabling the exit program using the OPENAPI option on the ENABLE PROGRAM command means that CICS overrides the CONCURRENCY setting on the exit's program definition with OPENAPI.) When CICS is connected to DB2 Version 6 or later, the CICS DB2 task-related user exit DFHD2EX1 is threadsafe. "SQL, threadsafe and other programming considerations for CICS DB2 applications" in the *CICS DB2 Guide* has more information on other exits that are particularly important for CICS DB2 requests. These exits include the default dynamic plan exit DSNCEXT (which is not defined as threadsafe), the alternative dynamic plan exit DFHD2PXT (which is defined as threadsafe), and the global user exits XRMIIIN and XRMIOU. Also be aware of the global user exits XEIIN and XEIOU, which are invoked before and after EXEC CICS commands, and XPCFTCH, which is invoked before a PPT-defined program receives control. Be sure that user exit programs supplied by any vendor software are coded to threadsafe standards and defined to CICS as threadsafe.
5. **If you are coding a user exit program** (a global user exit or a task-related user exit), you can define it as threadsafe so that it will be used on the same L8 TCB as a threadsafe application which calls it. Additionally, a task-related user exit can be enabled using the OPENAPI option on the ENABLE PROGRAM command so that it will be given control under an L8 TCB, use non-CICS APIs without having to create and manage subtask TCBs, and exploit the open transaction environment for itself. (Enabling the exit program using the OPENAPI option on the ENABLE PROGRAM command means that CICS overrides the CONCURRENCY setting on the exit's program definition with OPENAPI.) Global user exit programs can be handled in the same way as an ordinary application program—by using threadsafe application logic and threadsafe EXEC CICS commands, and defining the program as threadsafe. "Writing global user exit programs" in the *CICS Customization Guide* has general information about writing this type of program. For task-related user exit programs, see "Writing a task-related user exit program" in the *CICS Customization Guide* for more detailed information about how this type of program can exploit the open transaction environment safely. Note when you enable an exit program using the OPENAPI option, this indicates to CICS that the program's logic is threadsafe.

To make your program's application logic threadsafe, ensure that it uses appropriate serialization techniques when accessing shared resources. For most resources, such as files, transient data queues, temporary storage queues, and DB2 tables, CICS processing automatically ensures access in a threadsafe manner. As described above, some of the CICS commands that operate on these resources are coded to use appropriate serialization techniques that allow them to execute on open TCBs (that is, they are threadsafe commands). Where this is not the case, CICS ensures threadsafe processing by forcing a switch to the QR TCB, so that access to the resources is serialized regardless of the behaviour of the command. However, for any other resources which are accessed directly by user programs, such as shared storage, it is the responsibility of the user program to ensure threadsafe processing.

Typical examples of shared storage are the CICS CWA, the global work areas for global user exits, and storage acquired explicitly by the application program with the shared option. You can check whether your application programs use these types of shared storage by looking for occurrences of the following EXEC CICS commands:

- ADDRESS CWA
- EXTRACT EXIT
- GETMAIN SHARED

Although some of these commands are themselves threadsafe, they all give access to global storage areas, so the application logic that follows these commands and uses the global storage areas has the potential to be non-threadsafe. To ensure it is threadsafe, an application program must include the necessary synchronization logic to guard against concurrent update.

The load module scanner utility includes a sample table, DFHEIDTH, to help you identify whether the application logic in your existing application programs is threadsafe. DFHEIDTH contains the filter set for CICS commands that give access to shared storage. The use of these commands could make a program not threadsafe, unless it has the necessary synchronization logic in place to ensure serialization and prevent concurrent update.

Remember that DFHEIDTH, as supplied, is not testing the programs for non-threadsafe CICS commands (that is, commands that cause a switch to the QR TCB), but rather to determine if the application is using CICS commands which give rise to the possibility that the application logic could be not threadsafe. You can modify the supplied sample table to add any non-threadsafe CICS commands that you want to detect. Another sample table, DFHEIDNT, is supplied to help identify the CICS commands in your application which are non-threadsafe. For more information on using the load module scanner, see the *CICS Operations and Utilities Guide*.

Note: When identifying programs that use shared resources, you should also include any program that modifies itself. Such a program is effectively sharing storage and should be considered at risk.

Techniques that you can use to provide threadsafe processing when accessing a shared resource are as follows:

- Retry access, if the resource has been changed concurrently by another program, using the compare and swap instruction.
- Enqueue on the resource, to obtain exclusive control and ensure that no other program can access the resource, using:
 - An EXEC CICS ENQ command, in an application program
 - An XPI ENQUEUE function call to the CICS enqueue (NQ) domain, in a global user exit program
 - An MVS service such as ENQ (in an open API task-related user exit only when L8 TCBs are enabled for use). Note that the use of MVS services in an application which can execute under the QR TCB might result in performance degradation due to the TCB being placed in a wait.
- Perform accesses to shared resources only in a program that is defined as quasi-reentrant, by linking to the quasi-reentrant program using the EXEC CICS LINK command.

This technique applies to threadsafe application programs and open API task-related user exits only. A linked-to program defined as quasi-reentrant runs under the QR TCB and can take advantage of the serialization provided by CICS

quasi-reentrancy. Note that even in quasi-reentrant mode, serialization is provided only for as long as the program retains control and does not wait (see “Quasi-reentrant application programs” on page 147 for more information).

- Place all transactions that access the shared resource into a restricted transaction class (TRANCLASS), one that is defined with the number of active tasks specified as MAXACTIVE(1). This last approach effectively provides a very coarse locking mechanism, but may have a severe impact on performance.

Note: Although the term threadsafe is defined in the context of individual programs, a user application as a whole can only be considered threadsafe if all the application programs that access shared resources obey the rules. A program that is written correctly to threadsafe standards cannot safely update shared resources if another program that accesses the same resources does not obey the threadsafe rules.

Threadsafe considerations for statically or dynamically called routines

If you define a program with CONCURRENCY(THREADSAFE), all routines which are statically or dynamically called from that program (for example, Cobol routines) must also be coded to threadsafe standards.

When an EXEC CICS LINK command is used to link from one program to another, the program link stack level is incremented. However, a routine that is statically called, or dynamically called, does not involve passing through the CICS command level interface, and so does not cause the program link stack level to be incremented. With Cobol routines, for a static call, a simple branch and link is involved to an address that is resolved at linkedit time. For a dynamic call, although there is a program definition involved, this is required only to allow Language Environment to load the program. After that, a simple branch and link is executed. So when a routine is called by either of these methods, CICS does not regard this as a change of program. The program that called the routine is still considered to be executing, and so the program definition for that program is still considered to be the current one.

If the program definition for the calling program states CONCURRENCY(THREADSAFE), the called routine must also comply with this specification. Programs with the CONCURRENCY(THREADSAFE) attribute remain on an open TCB when they return from a DB2 call, and this is not appropriate for a program that is not threadsafe. For example, consider the situation where the initial program of a transaction, program A, issues a dynamic call to program B, which is a Cobol routine. Because the CICS command level interface was not involved, CICS is unaware of the call to program B, and considers the current program to be program A. Program B issues a DB2 call. On return from the DB2 call, CICS needs to determine whether the program can remain on the open TCB, or whether the program has to switch back to the QR TCB to ensure threadsafe processing. To do this, CICS examines the CONCURRENCY attribute of what it considers to be the current program, which is program A. If program A is defined as CONCURRENCY(THREADSAFE), then CICS allows processing to continue on the open TCB. In fact program B is executing, so if processing is to continue safely, program B must be coded to threadsafe standards.

OPENAPI programs

OPENAPI programs (that is, those defined in the resource definition with the OPENAPI attribute) are invoked on an open L8 or L9 mode TCB (depending on the EXECKEY attribute) instead of the main CICS QR TCB.

Obligations of OPENAPI programs

An OPENAPI program, although freed from the constraints imposed by the QR TCB, nevertheless does have obligations both to the CICS system as a whole and to future users of the L8 or L9 TCB it is using. An L8 or L9 TCB is dedicated for use by the CICS task to which it is allocated, but once the CICS task has completed, the TCB is returned to the dispatcher-managed pool of such TCBs, provided it is still in a "clean " state. (An unclean TCB in this context means that the task using the L8 or L9 mode TCB suffered an unhandled abend in an OPENAPI program. It does not mean that the program has broken the threadsafe restrictions, which CICS would not detect.) Note that the TCB is not dedicated for use by a particular OPENAPI program, but is used by all OPENAPI programs and OPENAPI TRUEs invoked by the CICS task to which the L8 mode TCB is allocated. Also, if an application program invoking an OPENAPI program is coded to threadsafe standards, and defined to CICS as threadsafe, it continues to execute on the L8 mode TCB on return from the program.

Threadsafe restrictions: An OPENAPI program must not treat the executing open TCB environment in such a way that it causes problems for:

- Application program logic that could run on the open TCB
- OPENAPI TRUEs called by the same task
- Future tasks that might use the open TCB
- CICS management code.

At your own risk, if your OPENAPI program decides to use other (non CICS) APIs, you must be aware of the following:

- When invoking CICS services, or when returning to CICS, an OPENAPI program must ensure it restores the MVS programming environment as it was on entry to the program. This includes cross-memory mode, ASC mode, request block (RB) level, linkage stack level, TCB dispatching priority, in addition to cancelling any ESTAEs added.
- At CICS task termination, an OPENAPI program must ensure it leaves the open TCB in a state suitable to be reused by another CICS transaction. In particular, it must ensure that all non-CICS resources acquired specifically on behalf of the terminating task are freed. Such resources might include:
 - Dynamically allocated data sets
 - Open ACBs or DCBs
 - STIMERM requests
 - MVS managed storage
 - ENQ requests
 - Attached subtasks
 - Loaded modules
 - Owned data spaces
 - Added access list entries
 - Name/token pairs
 - Fixed pages
 - Security settings (TCBSENV must be set to zero)
- An OPENAPI program must not use the following MVS system services that will affect overall CICS operation:
 - CHKPT
 - ESPIE
 - QEDIT

- SPIE
- STIMER
- TTIMER
- XCTL / XCTLX
- Any TSO/E services.
- An OPENAPI program must not invoke under the L8 or L9 mode TCB a Language Environment program that is using MVS Language Environment services, because L8 and L9 mode TCBs are initialized for Language Environment using CICS services.

Using the FORCEQR system initialization parameter

If you are running applications with programs defined as threadsafe to exploit OTE (for example, in CICS DB2 applications) problems could occur that indicate that one or more programs is not actually threadsafe. If this happens, you can force all your applications programs on to the QR TCB using the FORCEQR system initialization parameter. This could be particularly useful in a production region, where you cannot afford to have applications out of service while you investigate the problem.

The default for this parameter is FORCEQR=NO, which means that CICS honors the CONCURRENCY attribute on your program resource definitions. As a temporary measure, while you investigate and resolve problems connected with threadsafe-defined programs, you can set FORCEQR=YES. Remember to change this back to FORCEQR=NO when you are ready for your programs to resume use of open TCBs under the OTE.

Non-reentrant programs

There is nothing to prevent non-reentrant application programs being executed by CICS. However, such an application program would not provide consistent results in a multi-threading environment.

To use non-reentrant application programs, or tables or control blocks that are modifiable by the execution of associated application programs, specify the RELOAD(YES) option on their resource definition. RELOAD(YES) results in a fresh copy of the program or module being loaded into storage for each request. This option ensures that multithreading tasks that access a non-reentrant program or table each work from their own copy of the program, and are unaffected by changes made to another version of the program by other concurrent tasks running in the CICS region.

For information about RELOAD(YES), see the *CICS Resource Definition Guide*.

CICS/ESA loads any program link-edited with the RENT attributes into a CICS read-only dynamic storage area (DSA). CICS uses the RDSA for RMODE(24) programs, and the ERDSA for RMODE(ANY) programs. By default, the storage for these DSAs is allocated from read-only key-0 protected storage, protecting any modules loaded into them from all except programs running in key-zero or supervisor state. (If CICS initializes with the RENTPGM=NOPROTECT system initialization parameter, it does not use read-only key-0 storage, and use CICS-key storage instead.)

If you want to execute a non-reentrant program or module, it must be loaded into a non-read-only DSA. The SDSA and ESDSA are user-key storage areas for non-reentrant user-key programs and modules.

Storing data within a transaction

CICS provides a variety of facilities for storing data within and between transactions. Each one differs according to how available it leaves data to other programs within a transaction and to other transactions; in the way it is implemented; and in its overhead, recovery, and enqueueing characteristics.

Storage facilities that exist for the lifetime of a transaction include:

- Transaction work area (TWA)
- User storage (by a GETMAIN command issued without the SHARED option)
- COMMAREA
- Program storage

All of these areas are main storage facilities and come from the same basic source—the dynamic storage areas (DSAs) and extended dynamic storage areas (EDSAs). None of them is recoverable, and none can be protected by resource security keys. They differ, however, in accessibility and duration, and therefore each meets a different set of storage needs.

Transaction work area (TWA)

The transaction work area (TWA) is allocated when a transaction is initiated, and is initialized to binary zeroes. It lasts for the entire duration of the transaction, and is accessible to all local programs in the transaction. Any remote programs that are linked by a distributed program link command do not have access to the TWA of the client transaction. The size of the TWA is determined by the TWASIZE option on the transaction resource definition. If this size is nonzero, the TWA is always allocated. See the *CICS Resource Definition Guide* for more information about determining the TWASIZE.

Processor overhead associated with using the TWA is minimal. You do not need a GETMAIN command to access it, and you address it using a single ADDRESS command. The TASKDATAKEY option governs whether the TWA is obtained in CICS-key or user-key storage. (See Chapter 46, “Storage control,” on page 577 for a full explanation of CICS-key and user-key storage.) The TASKDATALOC option of the transaction definition governs whether the acquired storage can be above the 16MB line or not.

The TWA is suitable for quite small data storage requirements and for larger requirements that are both relatively fixed in size and are used more or less for the duration of the transaction. Because the TWA exists for the entire transaction, a large TWA size has much greater effect for conversational than for nonconversational transactions.

User storage

User storage is available to all the programs in a transaction, but some effort is required to pass it between programs using LINK or XCTL commands. Its size is not fixed, and it can be obtained (using GETMAIN commands) just when the transaction requires it and returned as soon as it is not needed. Therefore, user storage is useful for large storage requirements that are variable in size or are shorter-lived than the transaction.

See Chapter 46, “Storage control,” on page 577 for information about how USERDATAKEY and CICS DATAKEY override the TASKDATAKEY option of the GETMAIN command.

The SHARED option of the GETMAIN command causes the acquired storage to be retained after the end of the task. The storage can be passed in the communication area from one task to the next at the same terminal. The first task returns the address of the communication area in the COMMAREA option of the RETURN command. The second task accesses the address in the COMMAREA option of the ADDRESS command. You must use the SHARED option of the GETMAIN command to ensure that your storage is in common storage.

The amount of processor overhead involved in a GETMAIN command means that you should not use it for a small amount of storage. You should use the TWA for the smaller amounts or group them together into a larger request. Although the storage acquired by a GETMAIN command may be held somewhat longer when using combined requests, the processor overhead and the reference set size are both reduced.

COMMAREA in LINK and XCTL commands

A communication area (COMMAREA) is a facility used to transfer information between two programs within a transaction or between two transactions from the same terminal. For information about using COMMAREA between transactions, see “Using the COMMAREA in RETURN commands” on page 187.

Information in COMMAREA is available only to the two participating programs, unless those programs take explicit steps to make the data available to other programs that may be invoked later in the transaction. When one program links to another, the COMMAREA may be any data area to which the linking program has access. It is often in the working storage or LINKAGE SECTION of that program. In this area, the linking program can both pass data to the program it is invoking and receive results from that program.

When a program transfers control (an XCTL command) to another, CICS may copy the specified COMMAREA into a new area of storage, because the invoking program and its control blocks may no longer be available after it transfers control. In either case, the address of the area is passed to the program that is receiving control, and the CICS command-level interface sets up addressability. See Chapter 45, “Program control,” on page 565 for further information. When XCTL is used, CICS ensures that any COMMAREA is addressable by the program that receives it, by copying it below the 16MB line.

The COMMAREA is copied to USERKEY storage where necessary, depending on the addressing mode and EXECKEY attributes of the receiving program. See Chapter 46, “Storage control,” on page 577 for more information about EXECKEY. EXECKEY

CICS contains algorithms designed to reduce the number of bytes to be transmitted. The algorithms remove some trailing binary zeros from the COMMAREA before transmission and restore them after transmission. The operation of these algorithms is transparent to the application programs, which always see the full-size COMMAREA.

The overhead for using COMMAREA in an LINK command is minimal; it is slightly more with the XCTL and RETURN commands, when CICS creates the COMMAREA from a larger area of storage used by the program.

Channels in LINK and XCTL commands

Instead of using a communication area (COMMAREA), a more modern method of transferring data between CICS programs is to use a **channel**. Channels have several advantages over COMMAREAs—see Benefits of channels. To pass a channel on a LINK or XCTL command, you use the CHANNEL option in place of the COMMAREA option.

Channels are described in Enhanced inter-program data transfer: channels as modern-day COMMAREAs.

Program storage

CICS creates a separate copy of the variable area of a CICS program for each transaction using the program. This area is known as **program storage**. This area is called the WORKING-STORAGE SECTION in COBOL, automatic storage in C, C++, and PL/I, and the DFHEISTG section in assembler language. Like the TWA, this area is of fixed size and is allocated by CICS without you having to issue a GETMAIN command. The EXEC CICS interface sets up addressability automatically. Unlike the TWA, however, this storage lasts only while the program is being run, not for the duration of the transaction. This makes it useful for data areas that are not required outside the program and that are either small or, if large, are fixed in size and are required for all or most of the execution time of the program.

Temporary storage

Temporary storage is the primary CICS facility for storing data that must be available to multiple transactions.

Data items in temporary storage are kept in queues whose names are assigned dynamically by the program storing the data. A temporary storage queue containing multiple items can be thought of as a small data set whose records can be addressed either sequentially or directly, by item number. If a queue contains only a single item, it can be thought of as a named scratch-pad area.

Temporary storage data sharing means that main or auxiliary storage can be replaced by one or more temporary storage pools.

Temporary storage is implemented by the following methods:

- By using a particular queue that is determined by what is specified on the command that creates the first item
- By specifying the MAIN option so that the queue is kept in main storage, in space taken from the dynamic storage area
- By using the AUXILIARY option so that the queue is written to an entry-sequenced VSAM data set

Whichever method you use, CICS maintains an index of items in main storage.

Note that if the QNAME option matches the prefix of an installed TSMODEL resource definition, the MAIN or AUXILIARY value specified in the TSMODEL takes precedence over that specified in the command.

See the *CICS Resource Definition Guide* for more information about the use of TSMODELS to define temporary storage queues.

The addition of temporary storage data sharing gives another type of temporary storage queue that can be supported concurrently. These temporary storage queues can be defined as local, remote, or shared, and they can be stored in TS pools in the coupling facility.

These methods have characteristics that you should bear in mind:

- Main temporary storage requires much more virtual storage than auxiliary. In general, you should use it only for small queues that have short lifetimes or are accessed frequently. Auxiliary temporary storage is specifically designed for relatively large amounts of data that have a relatively long lifetime or are accessed infrequently. You may find it useful to establish a cutoff point of a lifetime of one second to decide which queues should be in main storage and which should be in auxiliary.
- You can make queues in auxiliary storage recoverable, but not queues in main storage:
- Shared temporary storage applies only to non-recoverable queues.
 - Only one transaction at a time can update a recoverable temporary storage queue. So, if you choose to make queues recoverable, bear in mind the probability of enqueues.
 - You should ensure that there are enough buffers and VSAM strings to eliminate as much contention as possible.
- If a task tries to write to temporary storage and there is no space available, CICS normally suspends it, although the task can regain control in this situation by using either a HANDLE CONDITION NOSPACE command, or the RESP or NOHANDLE option on the WRITEQ TS command. If suspended, the task is not resumed until some other task frees the necessary space in main storage or the VSAM data set. This can produce unexplained response delays, especially if the waiting task owns exclusive-use resources, in which case all other tasks needing those resources must also wait.
- It can be more efficient to use main temporary storage exclusively in very low-volume systems that have no need for recovery. You need to balance the needs for additional main storage requirement for the VSAM access method and a larger temporary storage program with the need for main storage for the temporary storage records.

The following points apply to temporary storage in general:

- You must use an EXEC CICS command every time data is written to or read from a temporary storage queue, and CICS must find or insert the data using its internal index. This means that the overhead for using main temporary storage is greater than for the CWA or TCTUA. With auxiliary storage, (often the most frequently used), there is usually data set I/O as well, which increases overhead even more.
- You need not allocate temporary storage until it is required; you need keep it only as long as it is required, and the item size is not fixed until you issue the command that creates it. This makes it a good choice for relatively high-volume data and data that varies in length or duration.
- The fact that temporary storage queues can be named as they are created provides a very powerful form of direct access to saved data. You can access scratch-pad areas for terminals, data set records, and so on, simply by including the terminal name or record key in the queue name.

- Resource protection is available for temporary storage.

Intrapartition transient data

Intrapartition transient data has some characteristics in common with auxiliary temporary storage. (See “Efficient sequential data set access” on page 175 for information about **extrapartition** transient data.) Like temporary storage, intrapartition transient data consists of queues of data, kept together in a single data set, with an index that CICS maintains in main storage.

You can use transient data for many of the purposes for which you would use auxiliary temporary storage, but there are some important differences.

- Transient data does not have the same dynamic characteristics as temporary storage. Unlike temporary storage queues, transient data queues cannot be created at the time data is written by an application program. However, transient data queues can be defined and installed using RDO while CICS is running.
- Transient data queues must be read sequentially. Each item can be read only once. After a transaction reads an item, that item is removed from the queue and is not available to any other transaction. In contrast, items in temporary storage queues may be read either sequentially or directly (by item number). They can be read any number of times and are not removed from the queue until the entire queue is purged.

These two characteristics make transient data inappropriate for scratch-pad data but suitable for queued data such as audit trails and output to be printed. In fact, for data that is read sequentially once, transient data is preferable to temporary storage.

- Items in a temporary storage queue can be changed; items in transient data queues cannot.
- Transient data queues are always written to a data set. (There is no form of transient data that corresponds to main temporary storage.)
- You can define transient data queues so that writing items to the queue causes a specific transaction to be initiated (for example, to process the queue). Temporary storage has nothing that corresponds to this “trigger” mechanism, although you may be able to use a START command to perform a similar function.
- Transient data has more recovery options than temporary storage. Transient data queues can be physically or logically recoverable.
- Because the commands for intrapartition and extrapartition transient data are identical, you can switch between the two types of data set. To do this, change only the transient data queue definitions and not your application programs themselves. Temporary storage has no corresponding function of this kind.

GETMAIN SHARED command

Storage acquired using the SHARED option of the GETMAIN command is not released when the acquiring task ends. This enables one task to leave data in storage for use by another task. The storage is not released until a FREEMAIN command is issued, either by the acquiring task or by another task.

Your own data sets

You can also use your own data sets to save data between transactions. This method probably has the largest overhead in terms of instructions processed, buffers, control blocks, and user programming requirements, but does provide extra functions and flexibility. Not only can you define data sets as recoverable resources,

but you can log changes to them for forward recovery. You can specify the number of strings for the data set, (as well as on the temporary storage and transient data sets), to ensure against access contention, and you can use resource security.

Lengths of areas passed to CICS commands

When a CICS command includes a LENGTH option, it usually accepts the length as a signed halfword binary value. This places a theoretical upper limit of 32KB on the length. In practice, the limits are less than this and vary for each command. The limits depend on data set definitions, recoverability requirements, buffer sizes, and local networking characteristics.

LENGTH options

In COBOL, C, C++, PL/I, and assembler language, the translator deals with lengths. See the *CICS Application Programming Reference* manual for programming information, including details of when you need to specify the LENGTH option. You should not let the length specified in CICS command options exceed 24KB, if possible.

Many commands involve the transfer of data between the application program and CICS. In all cases, the length of the data to be transferred must be provided by the application program.

In most cases, the LENGTH option must be specified if the SET option is used; the syntax of each command and its associated options show whether this rule applies.

There are options on the WAIT EXTERNAL command and a number of QUERY SECURITY commands that give the resource status or definition. CICS supplies the values associated with these options, hence the name, CICS-value data areas. The options are shown in the syntax of the commands with the term “cvda” in parentheses. For programming information about CVDAs, see the *CICS Application Programming Reference* manual.

For journal commands, the restrictions apply to the sum of the LENGTH and PFXLENG values. (See “Journaling” on page 255.)

Journal records

For journal records, the journal buffer size may impose a limit lower than 64KB. Note that the limit applies to the sum of the LENGTH and PFXLENG values.

Data set definitions

For temporary storage, transient data, and file control, the data set definitions can impose limits lower than 24KB. For details, see the *CICS System Definition Guide* (for information about creating data sets) and the *CICS Resource Definition Guide* (for information about resource definition for files).

Recommendation

For any command in any system, 32,000 bytes is a good working limit for LENGTH specifications. Subject to user-specified record and buffer sizes, this limit is unlikely either to cause an error or to place a constraint on applications.

You will probably not find a this limit too much of a hindrance; online programs do not often handle such large amounts of data, for the sake of efficiency and response time.

Note: The value in the LENGTH option must never exceed the length of the data area addressed by the command.

Minimizing errors

This section describes ways of making your applications error-free. Some of these suggestions apply not only to programming, but also to operations and systems.

What often happens is that, when two application systems that run perfectly by themselves are run together, performance goes down and you begin experiencing “lockouts” or waits. The scope of each system has not been defined well enough.

The key points in a well-designed application system are:

- At all levels, each function is defined clearly with inputs and outputs well-stated
- Resources that the system uses are adequately-defined
- Interactions with other systems are known

Protecting CICS from application errors

There are various tools and techniques you can use to minimize errors in your application programs. In general:

- You can use the storage protection facility to prevent CICS code and control blocks from being overwritten accidentally by your application programs. You can choose whether you want to use this facility by means of CICS system initialization parameters. See the *CICS System Definition Guide* for more information about this facility.
- Consider using standards that avoid problems that may be caused by techniques such as the use of GETMAIN commands.

Testing applications

The following general rules apply to testing applications:

- Do not test on a production CICS system—use a test system, where you can isolate errors without affecting “live” databases.
- Have the testing done by someone other than the application developer, if possible.
- Document the data you use for testing.
- Test your applications several times. See Chapter 49, “Testing applications,” on page 603 for more information about testing applications.
- Use the CEDF transaction for initial testing. See Chapter 50, “Execution diagnostic facility (EDF),” on page 607 for more information about using CEDF.
- Use stress or volume testing to catch problems that may not arise in a single-user environment. Teleprocessing Network Simulator (TPNS, licensed program number 5740-XT4) is a good tool for doing this.

TPNS is a telecommunications testing package that enables you to test and evaluate application programs before you install them. You can use TPNS for testing logic, user exit routines, message logging, data encryption, and device-dependencies, if these are used in application programs in your organization. It is useful in investigating system performance and response times, stress testing, and evaluating TP network design. For further information, see the *TPNS General Information* manual.

- Test whether the application can handle **correct** data and **incorrect** data.
- Test against complete copies of the related databases.
- Consider using multiregion operation. (See the *CICS Intercommunication Guide* for more information.)
- Before you move an application to the production system, it is a good idea to run a **final** set of tests against a copy of the production database to catch any errors.

In particular, look for destroyed storage chains.

Assembler language programs (if not addressing data areas properly) can be harder to identify because they can alter something that affects (and abends) another transaction.

For more information about solving a problem, see the *CICS Problem Determination Guide*.

Non-terminal transaction security

CICS can now protect, against unauthorized use, resources used in transactions that are not associated with a terminal. These transactions are of three types:

- Transactions that are started by a START command and that do not specify a terminal ID.
- Transactions that are started, without a terminal, as a result of the trigger level being reached for an intrapartition transient data queue.
- The CICS internal transaction (CPLT), which runs during CICS startup, to execute programs specified in the program list table (PLT). This transaction executes both first and second phases of PLTs.

Also, resource security checking can now be carried out for PLT programs that are run during CICS shutdown. PLT shutdown programs execute as part of the transaction that requests the shutdown, and therefore run under the authorization of the user issuing the shutdown command.

The START command handles security for non-terminal transactions started by the START command.

A surrogate user who is authorized to attach a transaction for another user, or cause it to be attached, or who inherits all the resource access authorizations for that transaction, can act for the user.

CICS can issue up to three surrogate user security checks on a single START command, depending on the circumstances:

1. The userid of the transaction that issues the START command, if USERID is specified
2. The userid of the CEDF transaction, if the transaction that issues the START command is being run in CEDF dual-screen mode
3. The CICS region userid of the remote system, if the START command is function shipped to another CICS system and link security is in effect.

A separate surrogate user security check is done for each of these userids, as required, before the transaction is attached.

For programming information about the USERID option, USERIDERR condition, and INVREQ, and NOTAUTH conditions, see the *CICS Application Programming Reference* manual.

Chapter 12. Design for performance

In this chapter, design changes are suggested that can improve performance and efficiency without much change to the application program itself.

- “Program size”
- “Virtual storage” on page 166
- “Exclusive control of resources” on page 169
- “Operational control” on page 170
- “Operating system waits” on page 171
- “The NOSUSPEND option” on page 171
- “Efficient data operations” on page 172
- “Efficient terminal operations” on page 176

Other aspects of application design are addressed in Chapter 11, “Application design,” on page 139

If you have a performance problem that applies in a particular situation, try to isolate the changes you make so that their effects apply only in that situation. After fixing the problem and testing the changes, use them in your most commonly-used programs and transactions, where the effects on performance are most noticeable.

Program size

The early emphasis on small programs led CICS programmers to break up programs into units that were as small as possible, and to transfer control using the XCTL command, or link using the LINK command, between them. In current systems, however, it is not always better to break up programs into such small units, because there is CICS processing overhead for each transfer and, for LINK commands, there is also storage overhead for the register save areas (RSAs).

For modestly-sized blocks of code that are processed sequentially, inline code is most efficient. The exceptions to this rule are blocks of code that are:

- Fairly long and used independently at several different points in the application
- Subject to frequent change (in which case, you balance the overhead of LINK or XCTL commands with ease of maintenance)
- Infrequently used, such as error recovery logic and code to handle uncommon data combinations

If you have a block of code that for one of these reasons, has to be written as a subroutine, the best way of dealing with this from a performance viewpoint is to use a closed subroutine within the invoking program (for example, code that is dealt with by a PERFORM command in COBOL). If it is needed by other programs, it should be a separate program. A separate program can be called, with a CALL statement (macro), or it can be kept separate and processed using an XCTL or a LINK command. Execution overhead is least for a CALL, because no CICS services are invoked; for example, the working storage of the program being called is **not** copied. A called program, however, must be linked into the calling one and so cannot be shared by other programs that need it unless you use special COBOL, C, or PL/I facilities. A called subroutine is loaded as part of each program that CALLs it and hence uses more storage. Thus, subsequent transactions using the program may or may not have the changes in the working storage made to the called program. This depends entirely on whether CICS has loaded a new copy of the program into storage.

Overhead (but also flexibility) is highest with the XCTL and LINK commands. Both processor and storage requirements are much greater for a LINK command than for an XCTL command. Therefore, if the invoking program does not need to have control returned to it after the invoked program is processed, it should use an XCTL command.

The load module resulting from any application program can occupy up to two gigabytes of main storage. Clearly, there is an extra cost associated with loading and initializing very large load modules, and CICS dynamic storage limits (EDSA) would need to be set correspondingly high. You should, if possible, avoid the use of large load modules. However large applications written in an object-oriented language, such as C++, can easily exceed 16M in size. Experience with C++ classes bound into a single DLL is that performance of the classes is degraded if the single DLL is reorganized into two or more DLLs. This is due to the processing required to resolve function references between multiple DLLs.

You may get an abend code of APCG if your program occupies all the available storage in the dynamic storage area (DSA).

Virtual storage

A truly conversational CICS task is one that converses with the terminal user for several or many interactions, by issuing a terminal read request after each write (for example, using either a SEND command followed by a RECEIVE command, or a CONVERSE command). This means that the task spends most of its extended life waiting for the next input from the terminal user.

Any CICS task requires some virtual storage throughout its life and, in a conversational task, some of this virtual storage is carried over the periods when the task is waiting for terminal I/O. The storage areas involved include the TCA and associated task control blocks (including EIS or EIB) and the storage required for all programs that are in use when any terminal read request is issued. Also included are the work areas (such as copies of COBOL working storage) associated with this task's use of those programs.

With careful design, you can sometimes arrange for only one very small program to be retained during the period of the conversation. The storage needed could be shared by other users. You must multiply the rest of the virtual storage requirement by the number of concurrent conversational sessions using that code.

By contrast, a pseudoconversational sequence of tasks requires almost all of its virtual storage only for the period actually spent processing message pairs. Typically, this takes a period of 1-3 seconds in each minute (the rest being time waiting for operator input). The overall requirement for multiple concurrent users is thus perhaps five percent of that needed for conversational tasks. However, you should allow for data areas that are passed from each task to the next. This may be a COMMAREA of a few bytes or a large area of temporary storage. If it is the latter, you are normally recommended to use temporary storage on disk rather than in main storage, but that means adding extra temporary storage I/O overhead in a pseudoconversational setup, which you do not need with conversational processing.

The extra virtual storage you need for conversational applications usually means that you need a correspondingly greater amount of real storage. The paging you need to control storage involves additional overhead and virtual storage. The

adverse effects of paging increase as transaction rates go up, and so you should minimize its use as much as possible. See Reducing paging effects for information about doing so.

Reducing paging effects

Reducing paging effects is a technique used by CICS in a virtual-storage environment. The key objective of programming in this environment is the reduction of page faults. A page fault occurs when a program refers to instructions or data that do not reside in real storage, in which case the page in virtual storage that contains the instructions or data referred to must be paged into real storage. The more paging required, the lower the overall system performance.

Although an application program may be able to communicate directly with the operating system, the results of such action are unpredictable and can degrade performance.

An understanding of the following terms is necessary for writing application programs to be run in a virtual-storage environment:

Locality of reference

The consistent reference, during the execution of the application program, to instructions and data within a relatively small number of pages (compared to the total number of pages in a program) for relatively long periods.

Working set

The number and combination of pages of a program needed during a given period.

Reference set

Direct reference to the required pages, without intermediate storage references that retrieve useless data.

Locality of reference

Keep the instructions processed and data used in a program within a relatively small number of pages (4096-byte segments). This quality in a program is known as “locality of reference”. You can do this by:

- Making the execution of the program as linear as possible.
- Keeping any subroutines you use in the normal execution sequence as close as possible to the code that invokes them.
- Placing code inline, even if you have to repeat it, if you have a short subroutine that is called from only a small number of places.
- Separating error-handling and other infrequently processed code from the main flow of the program.
- Separating data used by such code from data used in normal execution.
- Defining data items (especially arrays and other large structures) in the order in which they are referred to.
- Defining the elements within a data structure in the approximate order in which they are referred to. For example, in PL/I, all the elements of one row are stored, then the next row, and so on. You should define an array so that you can process it by row rather than by column.
- Initializing data as close as possible to where it is first used.
- Avoiding COBOL variable MOVE operations because these expand into subroutine calls.

- Issuing as few GETMAIN commands as possible. It is generally better for the program to add up its requirements and do one GETMAIN command than to do several smaller ones, unless the durations of these requirements vary greatly.
- Avoiding use of the INITIMG option on a GETMAIN command, if possible. It causes an immediate page reference to the storage that is obtained, which might otherwise not occur until much later in the program, when there are other references to the same area.

Note: Some of the suggestions above may conflict with your installation's programming standards if these are aimed at the readability and maintainability of the code, rather than speed of execution in a virtual-storage environment. Some structured programming methods, in particular modular programming techniques, make extensive use of the PERFORM command in COBOL (and the equivalent programming techniques in C, PL/I, and assembler language) to make the structure of the program clear. This may also result in more exceptions to sequential processing than are found in a nonstructured program. Nevertheless, the much greater productivity associated with structured code may be worth the possible loss of locality of reference.

Working set

The working set is the number and combination of pages of a program needed during a given period. To minimize the size of the working set, the amount of storage that a program refers to in a given period should be as small as possible. You can do this by:

- Writing modular programs and structuring the modules according to frequency and anticipated time of reference. Do not modularize merely for the sake of size; consider duplicate code inline as opposed to subroutines or separate modules.
- Using separate subprograms whenever the flow of the program suggests that execution is not be sequential.
- Not tying up main storage awaiting a reply from a terminal user.
- Using command-level file control locate-mode input/output rather than move-mode.
- In COBOL programs, specifying constants as literals in the PROCEDURE DIVISION, rather than as data variables in the WORKING STORAGE section.
- In C, C++, and PL/I programs, using static storage for constant data.
- Avoiding the use of LINK commands where possible, because they generate requests for main storage.

Reference set

Try to keep the overall number of pages that a program uses during normal operation as small as possible. These pages are termed the **reference set**, and they give an indication of the real storage requirement of the program. You can reduce the reference set by:

- Specifying constants in COBOL programs as literals in the PROCEDURE DIVISION, rather than as data variables in the WORKING STORAGE SECTION. The reason for this is that there is a separate copy of working storage for every task executing the program, whereas literals are considered part of the program itself, of which only one copy is used in CICS.
- Using static storage in C, C++, and PL/I for data that is genuinely constant, for the same reason as in the previous point.
- Reusing data areas in the program as much as possible. You can do this with the REDEFINES clause in COBOL, the union clause in C and C++, based storage in

PL/I, and ORG or equivalents in assembler language. In particular, if you have a map set that uses only one map at a time, code the DFHMSD map set definition without specifying either the STORAGE=AUTO or the BASE operand. This allows the maps in the map set to redefine one another.

Refer to data directly by:

- Avoiding long searches for data in tables
- Using data structures that can be addressed directly, such as arrays, rather than structures that must be searched, such as chains
- Avoiding methods that simulate indirect addressing

No attempt should be made to use overlays (paging techniques) in an application program. System paging is provided automatically and has superior performance. The design of an application program for a virtual-storage environment is similar to that for a real environment. The system should have all modules resident so that code on pages not referred to need not be paged in.

If the program is dynamic, the entire program must be loaded across adjacent pages before execution begins. Dynamic programs can be purged from storage if they are not being used and an unsatisfied storage request exists. Allowing sufficient dynamic area to prevent purging is more expensive than making them resident, because a dynamic program does not share unused space on a page with another program.

Exclusive control of resources

The very fundamental and powerful recovery facilities that CICS provides have performance implications. CICS serializes updates to recoverable resources so that if a transaction fails, its changes to those resources can be backed out independently of those made by any other transaction. Consequently, a transaction updating a recoverable resource gets control of that resource until it terminates or indicates that it wants to commit those changes with a SYNCPOINT command. Other transactions requiring the same resource must wait until the first transaction finishes with it.

The primary resources that produce these locking delays are data sets, DL/I databases, temporary storage, and transient data queues. The unit where protection is based is the individual record (key) for data sets, the program specification block (PSB) for DL/I databases, and the queue name for temporary storage. For transient data, the “read” end of the queue is considered a separate resource from the “write” end (that is, one transaction can read from a queue while another is writing to it).

To reduce transaction delays from contention for resource ownership, the length of time between the claiming of the resource and its release (the end of the UOW) should be minimized. In particular, conversational transactions should not own a critical resource across a terminal read.

Note: Even for nonrecoverable data sets, VSAM prevents two transactions from reading the same record for update at the same time. This enqueue ends as soon as the update is complete, however, rather than at the end of the UOW. Even this protection for a BDAM data set, can be relinquished by defining them with “no exclusive control” (SERVREQ=NOEXCTL) in the file control table.

This protection scheme can also produce deadlocks as well as delays, unless specific conventions are observed. If two transactions update more than one recoverable resource, they should always update the resources in the same order. If they each update two data sets, for example, data set “A” should be updated before data set “B” in all transactions. Similarly, if transactions update several records in a single data set, they should always do so in some predictable order (low key to high, or conversely). You might also consider including the TOKEN keyword with each READ UPDATE command. See “The TOKEN option” on page 306 for information about the TOKEN keyword. Transient data, temporary storage, and user journals must be included among such resources. The *CICS Recovery and Restart Guide* contains further information on the extent of resource protection.

It may be appropriate here to note the difference between CICS data sets on a VSAM control interval, and VSAM internal locks on the data set. Because CICS has no information about VSAM enqueue, a SHARE OPTION 4 control interval that is updated simultaneously from batch and CICS can result in, at best, reduced performance and, at worst, an undetectable deadlock situation between batch and CICS. You should avoid such simultaneous updates between batch and CICS. In any case, if a data set is updated by both batch and CICS, CICS is unable to ensure data integrity.

Operational control

The following operational techniques can be used to influence the performance and efficiency of the CICS system:

MXT

The CICS system initialization parameter MXT specifies the maximum number of user tasks that can exist in a CICS system at the same time. MXT is invaluable for avoiding short-on-storage (SOS) conditions and for controlling contention for resources in CICS systems. It works by delaying the creation of user tasks to process input messages, if there are already too many activities in the CICS system. In particular, the virtual storage occupied by a message awaiting processing is usually much less than that needed for the task to process it, so you save virtual storage by delaying the processing of the message until you can do so quickly.

Transaction classes are useful in limiting the number of tasks of a particular user-defined type, or class, if these are heavy resource users.

Runaway tasks

CICS only resets a task’s runaway time (ICVR) when a task is suspended. An EXEC CICS command cannot be guaranteed to cause a task to suspend during processing because of the unique nature of each CICS implementation. The runaway time may be exceeded causing a task to abend AICA. This abend can be prevented by coding an EXEC CICS SUSPEND command in the application. This causes the dispatcher to suspend the task that issued the request and allow any task of higher priority to run. If there is no task ready to run, the program that issued the suspend is resumed. For further information about abend AICA, see the *CICS Problem Determination Guide*.

Auxiliary trace

Use auxiliary trace to review your application programs. For example, it can show up any obviously unnecessary code, such as a data set browse from the beginning of a data set instead of after a SETL, too many or too large

GETMAIN commands, failure to release storage when it is no longer needed, unintentional logic loops, and failure to unlock records held for exclusive control that are no longer needed.

Operating system waits

You should avoid using facilities that cause operating system waits. All CICS activity stops when one of these waits occurs, and all transactions suffer response delays. The chief sources of such waits are:

- Extrapartition transient data sets. (See “Efficient sequential data set access” on page 175.)
- Those COBOL, C, C++, and PL/I language facilities that you should not use in CICS programs and for which CICS generally provides alternative facilities. For guidance information about the language restrictions, see Chapter 4, “Programming in COBOL,” on page 21, Chapter 5, “Programming in C and C++,” on page 43, and Chapter 6, “Programming in PL/I,” on page 53.
- SVCs and assembler language macros that invoke operating system services, such as write-to-operator (WTO).

The NOSUSPEND option

The default action for the ENQBUSY, NOJBUFSP, NOSPACE, NOSTG, QBUSY, SESSBUSY, and SYSBUSY conditions is to suspend the execution of the application until the required resource (for example, storage) becomes available, and then resume processing the command. The commands that can give rise to these conditions are: ALLOCATE, ENQ, GETMAIN, WRITE JOURNALNAME, WRITE JOURNALNUM, READQ TD, and WRITEQ TS.

On these commands, you can use the NOSUSPEND option (also known as the NOQUEUE option in the case of the ALLOCATE command) to inhibit this waiting and cause an immediate return to the instruction in the application program following the command.

CICS maintains a table of conditions referred to by the HANDLE CONDITION and IGNORE CONDITION commands in a COBOL application program¹. Execution of these commands either updates the existing entry, or causes a new entry to be made if the condition has not yet been the subject of such a command. Each entry indicates one of the three states described below:

- A label is currently specified, as follows:
`HANDLE CONDITION condition(label)`
- The condition is to be ignored, as follows:
`IGNORE CONDITION`
- No label is currently specified, as follows:
`HANDLE CONDITION`

1. HANDLE CONDITION and IGNORE CONDITION commands are not supported for C and C++ programs.

When the condition occurs, the following tests are made:

1. If the command has the NOHANDLE or RESP option, control returns to the next instruction in the application program. Otherwise, the condition table is scanned to see what to do.
2. If an entry for the condition exists, this determines the action.
3. If no entry exists and the default action for this condition is to suspend execution:
 - If the command has the NOSUSPEND or NOQUEUE option, control returns to the next instruction.
 - If the command does not have one of these options, the task is suspended.
4. If no entry exists and the default action for this condition is to abend, a second search is made looking for the ERROR condition:
 - If found, this entry determines the action.
 - If ERROR is searched for and not found, the task is abended.

Efficient data operations

CICS supports:

- DL/I database operations
- VSAM and BDAM data set operations
- Browsing
- Logging
- Sequential data set access

Efficient database operations

The following recommendations apply to using DL/I with CICS:

- Use command codes with CALL level and keywords with command level to reduce the number of requests whenever appropriate. See the *CICS IMS Database Control Guide* for more information. For example, a DL/I path call is more efficient than a number of individual DL/I calls. With individual DL/I calls, the GN call gives the best performance. Although several DL/I calls may get their information from the DL/I or VSAM buffers, some of the instructions have to be processed within a DL/I call. You should, therefore, consider the number of DL/I calls needed for the processing of a transaction.
- It is more efficient to use qualified segment-search areas (SSAs) than to check on “segment found” in the application program.
- Scheduling calls should be issued at the latest possible time, so as to minimize the time that the transaction has exclusive control of the PSB. (This control is released at the end of the UOW, which occurs at the next TERM call, explicit SYNCPOINT command, or the syncpoint implicit in task termination.)
- Be aware of the effects of explicit syncpointing on performance and recovery.

Efficient data set operations

The efficiency of database and data set operations is an important factor in the performance of any CICS system. In VSAM, the main impact on efficiency, and thus on response time, comes from contention for serial-use resources (record keys, control intervals, and strings), and for storage use and processor overhead. As is usual in these situations, any improvements you make in one area may be at the expense of other areas.

VSAM data sets

To minimize contention delays using VSAM data sets:

- Minimize the time that VSAM resources are reserved for exclusive use. The **exclusive use** enqueue is the way CICS and VSAM prevent concurrent updates.

If you use VSAM record-level sharing, described in “Accessing files in RLS mode” on page 289, VSAM locks a record that has been requested for update, so that no other transaction can attempt to update the record at the same time. If the file is recoverable, VSAM releases the lock at the next syncpoint. If the file is not recoverable, VSAM releases the lock when the request is complete. The recoverability of a file is defined in the integrated catalog facility (ICF) catalog.

If you do not use VSAM record-level sharing, CICS serializes update requests by base cluster record key. The complete VSAM control interval (CI) containing the requested record is held for exclusive use while an individual command (for example, a READ command with the UPDATE option) is being executed on the record. Once each command is complete, the control interval is released, and only the requested record remains locked. For **nonrecoverable** data sets, both the VSAM exclusive use and the CICS exclusive use of the record end when the update request is complete in VSAM terms; for example, when the REWRITE command has completed. For **recoverable data sets**, however, the CICS exclusive use does not end until the task ends or issues a SYNCPOINT command. Recoverability is specified in the data set resource definition. See the *CICS Resource Definition Guide* for more information about the FILE resource definitions.

- Hold position in a VSAM data set for as short a time as possible. Table 8 shows which commands hold position and when the hold is released.

Table 8. Commands that hold position and when hold is released

Command	Released by VSAM at
READ.. UPDATE	REWRITE/DELETE/UNLOCK
WRITE.. MASSINSERT	UNLOCK
STARTBR	ENDBR

Each request in progress against a VSAM data set requires at least one string. Requests that hold position tie up a string until a command is issued to release the hold position. Requests that do not hold position release the string as soon as that request is complete.

To minimize processor overhead when using VSAM data sets:

- Use the MASSINSERT option if you are adding many records in sequence. This improves performance by minimizing processor overheads and therefore improves the response times. For ESDSs and KSDSs, adding records with MASSINSERT causes CICS to use sequential VSAM processing. This changes the way VSAM places records within control intervals when a split is required, resulting in fewer splits and less unused space within the affected CIs.
- Use **skip sequential** processing if you are reading many records in sequence whose keys are relatively close together but not necessarily adjacent. (Skip sequential processing begins with a start browse (STARTBTR command).) Each record is retrieved with an READNEXT command, but the key feedback area pointed to by RIDFLD is supplied with the key of the next requested record before the READNEXT command is issued.
- Use the GENERIC option on the DELETE command when deleting a group of records whose keys start with a common character string. CICS internally optimizes a generic DELETE.

BDAM data sets

BDAM data sets are less efficient than VSAM because CICS has to do some single-thread processing and issue some operating system waits to handle BDAM data set requests. Therefore, if possible, you should use a relative record VSAM data set or an entry-sequenced data set addressed by relative byte address (RBA) in place of a BDAM data set.

If you are using BDAM data sets in update mode, you should be aware that performance is affected dramatically by the means of data set integrity you choose.

If you specify **exclusive control** in file control table SERVREQ operands for a BDAM data set, CICS requests the operating system to prevent concurrent updates. However, this involves significant overhead.

Efficient browsing (in non-RLS mode)

A data set browse is often the source of the output in transactions that produce a large number of output screens, which can monopolize system resources. A long browse can put a severe load on the system, locking out other transactions and increasing overall response time, in addition to the overhead needed for BMS, task control, and terminals. This is because CICS control philosophy is based on the assumption that the terminal operator initiates a transaction that accesses a few data records, processes the information, and returns the results to the operator. This process involves numerous waits that enable CICS to do multitasking. However, CICS is not an interrupt-driven multitasking system, so tasks that involve small amounts of I/O relative to processing can monopolize the system regardless of priority. A browse of a data set with many records in a control interval is just such a transaction.

You can prevent this by issuing DELAY or SUSPEND commands periodically, so that other tasks can get control. If the browse produces paged output, you should consider breaking the transaction up in one of the ways suggested in “Page-building and routing operations” on page 179.

Efficient logging

CICS provides options to log some or all types of activity against a data set. Logging updates enables you to reconstruct data sets from backup copies, if necessary. You may also want to log reads for security reasons. Again, you have to balance the need for data integrity and security against the performance effects of logging. These are the actual operations needed to do the logging and the possible delays caused because of the exclusive control that logging implies.

Efficient sequential data set access

CICS provides a number of different sequential processing options. Temporary storage and intrapartition transient data queues (already discussed in “Temporary storage” on page 158 and in “Intrapartition transient data” on page 160) are the most efficient to use, but they must be created and processed entirely within CICS.

Extrapartition transient data is the CICS way of handling standard sequential (QSAM/BSAM) data sets. It is the least efficient of the three forms of sequential support listed, because CICS has to issue operating system waits to process the data sets, as it does when handling BDAM. Moreover, extrapartition transient data sets are not recoverable. VSAM ESDSs, on the other hand, are recoverable within limitations, and processing is more efficient. The recovery limitation is that records added to an ESDS during an uncompleted UOW cannot be removed physically during the backout process, because of VSAM restrictions. They can, however, be flagged as deleted by a user exit routine.

CICS journals provide another good alternative to extrapartition transient data, although only for output data sets. Journals are managed by the MVS system logger, but flexible processing options permit very efficient processing. Each journal command specifies operation characteristics, for example, synchronous or asynchronous, whereas extrapartition operations are governed entirely by the parameters in the transient data queue definition.

Transactions should journal asynchronously, if possible, to minimize task waits in connection with journaling. However, if integrity considerations require that the journal records be physically written before end of task, you must use a synchronous write. If there are several journal writes, the transaction should use asynchronous writes for all but the last logical record, so that the logical records for the task are written with a minimum number of physical I/Os and only one wait.

You can use journals for input (in batch) as well as output (online) while CICS is running. The supplied batch utility DFHJUP can be used for access to journal data, for example, by printing or copying. Note that reading a journal in batch involves the following restrictions:

- Access to MVS system logger log stream data is provided through a subsystem interface, LOGR.
- Reading records from a journal is possible offline by means of a batch job only.

Efficient terminal operations

There are some design factors, related to communicating with terminals, that may affect performance.

Length of the data stream sent to the terminal

Good screen design and effective use of 3270 hardware features can significantly affect the number of bytes transmitted on a teleprocessing link. It is particularly important to keep the number of bytes as small as possible because, in most cases, this is the slowest part of the path a transaction takes. The efficiency of the data stream therefore affects both response time and line usage.

Basic mapping support considerations

When building a formatted data stream with basic mapping support (BMS), you should bear in mind, the factors described in the following sections.

Avoid turning on modified data tags (MDTs) unnecessarily

The MDT is the bit in the attribute byte that determines whether a field should be transmitted on a READ MODIFIED command (the command used by CICS for all but copy operations).

The MDT for a field is normally turned on by the 3270 hardware when the user enters data into a field. However, you can also turn the tag on when you send a map to the screen, either by specifying FSET in the map or by sending an override attribute byte that has the tag on. You should never set the tag on in this way for a field that is constant in the map, or for a field that has no label (and is not sent to the program that receives the map).

Also, you do not normally need to specify FSET for an ordinary input field. This is because, as already mentioned, the MDT is turned on automatically in any field in which the user enters data. This is then included in the next RECEIVE command. These tags remain on, no matter how many times the screen is sent, until explicitly turned **off** by the program (by the FRSET, ERASEAUP, or ERASE option, or by an override attribute with the tag off).

You can store information, between inputs, that the user did not enter on the screen. This is an intended reason for turning the MDT on by a program. However, this storage technique is appropriate only to small amounts of data, and is more suitable for local than for remote terminals, because of the transmission overhead involved. For example, this technique is particularly useful for storing default values for input fields. In some applications, the user must complete a screen in which some fields already contain default values. A user who does not want to change a default just skips that field. The program processing the input has to be informed what these defaults are. If they are always the same, they can be supplied as constants in the program. If they are variable, however, and depend on earlier inputs, you can simply save them on the screen by turning the MDT on with FSET in the map that writes the screen. The program reading the screen then receives the default value from a user who does not change the field and the new value from a user who does.

Note: The saved values are not returned to the screen if the CLEAR, PA1, PA2, or PA3 key is pressed.

Use FRSET to reduce inbound traffic

If you have a screen with many input fields, which you may have to read several times, you can reduce the length of the input data stream by specifying FRSET when you write back to the screen in preparation for the next read. FRSET turns off the MDTs, so that fields entered before that write are not present unless the user reenters them the next time. If you are dealing with a relatively full screen and a process where there may be a number of error cycles (or repeat transmissions for some other reason), this can be a substantial saving. However, because only **changed** fields are sent on subsequent reads, the program must save input from each cycle and merge the new data with the old. This is not necessary if you are not using FRSET, because the MDTs remain on, and all fields are sent regardless of when they were entered.

Do not send blank fields to the screen

Sending fields to the screen that consist entirely of blanks or that are filled out on the right by trailing blanks usually wastes line capacity. The only case where BMS requires you to do this is when you need to erase a field on the screen that currently contains data, or to replace it with data shorter than that currently on the screen, without changing the rest of the screen.

This is because, when BMS builds the data stream representing your map, it includes blanks (X'40') but omits nulls (X'00'). This makes the output data stream shorter. BMS omits any field whose first data character is null, regardless of subsequent characters in the field.

BMS requires you to initialize to nulls any area to be used to build a map. This is done by moving nulls (X'00') to the mapnameO field in the symbolic map structure. See “Initializing the output map” on page 478 for more information. BMS uses nulls in attribute positions and in the first position of data to indicate that no change is to be made to the value in the map. If you are reusing a map area in a program or in a TIOA, you should take special care to clear it in this way.

Address CICS areas correctly

There are several ways to check that CICS areas are addressed correctly. Ensure that:

- Each COBOL program with a LINKAGE SECTION structure that exceeds 4KB has the required definition and the setting of more than one contiguous BLL cell.
- Every BLL pointer points to an area that is a 01-level item.
- Call level DL/I is only used with PSBs that are correctly addressed.

Use the MAPONLY option when possible

The MAPONLY option sends only the **constant** data in a map, and does not merge any variable data from the program. The resulting data stream is not always shorter, but the operation has a shorter path length in BMS. When you send a skeleton screen to be used for data entry, you can often use MAPONLY.

Send only changed fields to an existing screen

Sending only changed fields is important when, for example, a message is added to the screen, or one or two fields on an input screen are highlighted to show errors. In these situations, you should use the DATAONLY option to send a map that consists of nulls except for the changed fields. For fields where the only the attribute byte has changed, you need send only that byte, and send the remaining fields as nulls. BMS uses this input to build a data stream consisting of only the fields in question, and all other fields on the screen remain unchanged.

It may be tempting to ignore this advice and send an unnecessarily long data stream. For example, when a program that is checking an input screen for errors finds one, there are two options.

- It can simply add the error information to the input map (highlighted attributes, error messages, and so on) and resend it.
- It can build an entirely new screen, consisting of just the error and message fields.

The former is slightly easier to code (you do not need to have two map areas or move any fields), but it may result in very much longer transmissions because the output data stream contains the correct input fields as well as the error and message fields. In fact, it may even be longer than the original input stream because, if there were empty or short fields in the input, BMS may have replaced the missing characters with blanks or zeros.

With the 3270 hardware, if the input stream for a terminal exceeds 256 bytes, the terminal control unit automatically breaks it up into separate transmissions of 256 bytes maximum. This means that a long input stream may require several physical I/O operations. Although this is transparent to the application program, it does cause additional line and processor overhead. The **output** stream is generally sent in a single transmission.

Design data entry operations to reduce line traffic

Often, users are required to complete the same screen several times. Only the data changes on each cycle; the titles, field labels, instructions, and so on remain unchanged. In this situation, when an entry is accepted and processed, you can respond with a SEND CONTROL ERASEAUP command (or a map that contains only a short confirmation message and specifies the ERASEAUP option). This causes all the **unprotected** fields on the screen (that is, all the input data from the last entry) to be erased and to have their MDTs reset. The labels and other text, which are in protected fields, are unchanged, the screen is ready for the next data-entry cycle, and only the necessary data has been sent.

Compress data sent to the screen

When you send unformatted data to the screen, or create a formatted screen outside BMS, you can compress the data further by inserting set buffer address (SBA) and repeat-to-address (RA) orders into the data stream. SBA allows you to position data on the screen, and RA causes the character following it to be generated from the current point in the buffer until a specified ending address. SBA is useful whenever there are substantial unused areas on the screen that are followed by data. RA is useful when there are long sequences of the same character, such as blanks or dashes, on the screen. However, you should note that the speed with which RA processes is not uniform across all models of 3270 control units. You should check how it applies to your configuration before use.

CICS provides an exit that is driven just before output is sent to a terminal (XTC OUT). You may want to add SBA and RA substitutions to this exit to compress the data stream using a general subroutine. This has the dual benefit of removing compression logic from your application program and of applying to all output data streams, whether they are produced by BMS or not.

Use nulls instead of blanks

You should note that, outside BMS, nulls have no special significance in an **output** data stream. If you need a blank area on a screen, you can send either blanks or

nulls to it; they take up the same space in the output stream. However, if the blank field is likely to be changed by the user and subsequently read, use nulls, because they are not transmitted back.

Use methods that avoid the need for nulls or blanks

For any *large* area of a screen that needs to be blank, you should consider methods other than transmitting blanks or nulls; for example, when using BMS, putting SBA and RA orders directly into the data stream, or using the ERASE and ERASEAUP options.

Page-building and routing operations

BMS page-building facilities provide a powerful and flexible tool for building and displaying long messages, sending messages to multiple destinations, and formatting a single message for several devices with different physical characteristics. However, as for any high-function tool, it requires a substantial overhead, as mentioned in “Efficient browsing (in non-RLS mode)” on page 174. You may need the page-building option (ACCUM) when:

- Sending messages whose length exceeds the capacity of the output device (multipage output)
- Using destinations other than the input terminal
- Sending pages built from multiple maps
- Using the BMS page-copy facility

Sending multipage output

Transactions that produce very large output messages, consisting of many screen-size pages, tend to tax system resources. First, all the pages have to be created, which involves processor activity, execution of the CSPG transaction, and data set I/O activity. The pages must then be saved in temporary storage. If the terminal user looks at every page in a message, a large number of transactions are run to process the paging requests, each of which needs line and processor overhead. Obviously some overhead is caused by the size and complexity of the transaction, and it may be unavoidable. Indeed, if several users are scrolling rapidly through paged output at the same time, the transactions needed can monopolize a system.

If users really need to see all the pages, and need to scroll backward and forward frequently, it may be more efficient to produce all the pages at the same time and present them using “traditional” CICS paging services. However, if users need only a few of the pages, or can easily specify how far back or forward in the message they would like to scroll, there are two choices:

1. First, construct a pseudoconversational transaction to produce just one screen of output. The first time this transaction is run, it produces the first page of the many-page output. The output screen contains space for users to indicate the page they want next. The transaction always sets the next transaction identifier to point to itself, so that it can display the requested page when it is next run. You will probably want to give users some of the options that CICS provides (such as one page forward, one page back, and skip to a selected page) and some relevant to the application, such as a data set key at which to begin the next page of output.
2. The alternative is to page-build a multipage output message with the ACCUM option, but to limit the number of pages in the message (say to five). Users page through the subset pages with the usual CICS page commands. On the last screen of the output, you add an indication that there is more output and a place for them to indicate whether they want to see the next segment. As in the

first example, the next transaction identifier is set to the original transaction so that, if CICS does not receive a paging command, it invokes that transaction.

Sending messages to destinations other than the input terminal

If you need to send a message to a terminal other than the input terminal associated with a task, BMS routing may be the most efficient way of doing so. This is especially so if the message must be sent to multiple destinations or if it involves multiple pages. Routing is the recommended method if the message recipients need CICS paging commands to access it.

However, if neither of the above conditions apply, you have a choice of two other methods of delivering output to a terminal not associated with the transaction.

1. You can use a START command, with the TERMID option, to specify the terminal to which you want to write and the FROM option to specify the data you want to send. Your own transaction is the started transaction. It issues an RETRIEVE command for the message and then sends it to its own terminal. See the *CICS Application Programming Reference* manual for programming information about the START command.

2. Similarly, you can put messages destined for a particular terminal on to an intrapartition transient data queue. The definition for the transient data queue must specify:

- The destination as a TERMINAL
- The terminal identifier
- A trigger level
- A transaction name

Your own transaction reads the transient data queue and sends the message to its terminal. It repeats this sequence until the queue is empty, and then terminates. The trigger level you specified means that it is invoked every time the specified number of messages have been placed on the queue. The *CICS/ESA Sample Applications Guide* describes the DFHœTDWT sample program that performs this function.

Note: Because of the overhead associated with routing messages (by whatever means), you should use facilities such as ROUTE=ALL with caution.

Sending pages built from multiple maps

Although you can easily build a screen gradually using different maps, you can sometimes avoid considerable overhead by not using page-building operations, especially where there is only one screen of output and no other need for paging. An example of this is an application whose output consists of a header map, followed by a variable number of detail segments, sent with a second map, and finally a trailer map following the detail. Suppose the average output screen for such an application contains eight (2-line) detail segments, plus header and trailer, and all this fits on a single screen. Writing this screen with page-building requires 11 BMS calls (header, details, trailer, and page-out) whereas, if the program builds the output screen internally, it only needs one call.

Using the BMS page-copy facility

Because the individual pages that make up an accumulated BMS message are saved in temporary storage, BMS enables the terminal user to copy individual pages to other terminals. However, if the ability to copy is the only reason for using page-building, you should consider using either the 3274 control unit copy facilities or the CICS **copy key** facility instead.

The 3274 copy facilities require no participation from CICS and no transmission, and are by far the most efficient method. The CICS copy key facility does have an overhead (see “Requests for printed output”), although of a different type from the BMS copy facility. It also has destination restrictions that do not apply to BMS copying.

Requests for printed output

A CICS print request asks CICS to copy what is on the requesting screen to the first available printer on the same control unit. The overhead involved depends on whether a printer is available, and whether the requesting terminal is remote or local to CICS.

If no printer is available, and the request is from a remote or a local device:

- CICS reads the buffer to the display terminal. This involves transmitting **every** position on the screen, including nulls.

For requests from a local device, the READ BUFFER command takes place at channel speeds, so that the large input message size does not increase response time too much, and does not monopolize the line.

- An error task is generated so that the terminal error program can dispose of the message. If a printer is available and the request is from a local device, this step is not needed.
- The 3270 print task (CSPP) is attached to write the entire buffer to the printer when it is available.

If a printer is available, and the request is from a remote device, CICS sends a very short data stream to the control unit asking for a copy of the requesting device buffer to be sent to the output device buffer.

Additional terminal control considerations

The following sections describe additional points to consider when using the CICS terminal control services.

Use only one physical SEND command per screen

We mentioned earlier that it is usually more efficient to create a screen with a single call to BMS, than to build the screen with a series of SEND MAP ACCUM commands. It is important to send the screen in a single physical output to the terminal. It is **very** inefficient to build a screen in parts and send each part with a separate command, because of the additional processor overhead of using several commands and the additional line and access method overhead.

Use the CONVERSE command

Use the CONVERSE command rather than the SEND and RECEIVE commands (or a SEND, WAIT, RECEIVE command sequence if your program is conversational). They are functionally equivalent, but the CONVERSE command crosses the CICS services interface only once, which saves processor time.

Limit the use of message integrity options

Like specifying the WAIT option on the final SEND command of a transaction, the MSGINTEG option of CEDA requires CICS to keep the transaction running until the last message has been delivered successfully.

The PROTECT option of the PROFILE definition implies message integrity and causes the system to log all input and output messages, which adds to I/O and processor overhead.

Avoid using the DEFRESP option on SEND commands

Avoid using the DEFRESP option on SEND commands, unless the transaction must verify successful delivery of the output message. It delays termination of the transaction in the same way as MSGINTEG.

Avoid using unnecessary transactions

Avoid situations that may cause users to enter an incorrect transaction or to use the CLEAR key unnecessarily, thus adding to terminal input, task control processing, terminal output, and overhead. Good screen design and standardized PF and PA key assignments should minimize this.

Send unformatted data without maps

If your output to a terminal is entirely or even mostly unformatted, you can send it using terminal control commands rather than BMS commands (that is, using a BMS SEND command without the MAP or TEXT options).

Chapter 13. Sharing data across transactions

CICS facilities for sharing data across transactions include:

- The Common Work Area (CWA)
- The TCTTE user area (TCTUA)
- The COMMAREA
- The display screen
- Channels and containers

Data stored in the TCTUA and the CWA is available to any transaction in the system. Subject to resource security and storage protection restrictions, any transaction may write to them and any transaction may read them.

The use of some of these facilities may cause inter-transaction affinities. See Chapter 15, “Affinity,” on page 221 for more information about transaction affinities.

This chapter describes:

- “Using the common work area (CWA)”
- “Using the TCTTE user area (TCTUA)” on page 186
- “Using the COMMAREA in RETURN commands” on page 187
- “Using a channel on RETURN commands” on page 187
- “Using the display screen to share data” on page 188

Using the common work area (CWA)

The common work area (CWA) is a single control block that is allocated at system startup time and exists for the duration of that CICS session. The size is fixed, as specified in the system initialization parameter, WRKAREA. The CWA has the following characteristics:

- There is almost no overhead in storing or retrieving data from the CWA. Command-level programs must issue one ADDRESS command to get the address of the area but, after that, they can access it directly.
- Data in the CWA is not recovered if a transaction or the system fails.
- It is not subject to resource security.
- CICS does not regulate use of the CWA. All programs in all applications that use the CWA must follow the same rules for shared use. These are usually set down by the system programmers, in cooperation with application developers, and require all programs to use the same “copy” module to describe the layout of the area.

You must not exceed the length of the CWA, because this causes a storage violation. Furthermore, you must ensure that the data used in one transaction does not overlay data used in another. One way to protect CWA data is to use the storage protection facility that protects the CWA from being written to by user-key applications. See “Protecting the CWA” on page 184 for more information.

- The CWA is especially suitable for small amounts of data, such as status information, that are read or updated frequently by multiple programs in an application.
- The CWA is not suitable for large-volume or short-lived data because it is always allocated.

Protecting the CWA

The CWAKEY system initialization parameter allows you to specify whether the CWA is to be allocated from CICS-key or user-key storage. See the *CICS System Definition Guide* for details about the CWAKEY parameter.

If you want to restrict write access to the CWA, you can specify CWAKEY=CICS. This means that CICS allocates the CWA from CICS-key storage, restricting application programs defined with EXECKEY(USER) to read-only access to the CWA. The only programs allowed to write to a CWA allocated from CICS-key storage are those you define with EXECKEY(CICS).

Because any program that executes in CICS key can also write to CICS storage, you should ensure that such programs are thoroughly tested to make sure that they do not overwrite CICS storage.

If you want to give preference to protecting CICS rather than the CWA, specify CWAKEY=USER for the CWA, and EXECKEY(USER) for all programs that write to the CWA. This ensures that if a program exceeds the length of the CWA it does not overwrite CICS storage. For more information about storage protection, see Chapter 46, “Storage control,” on page 577.

Figure 39 illustrates a particular use of the CWA where the CWA itself is protected from user-key application programs by CWAKEY=CICS.

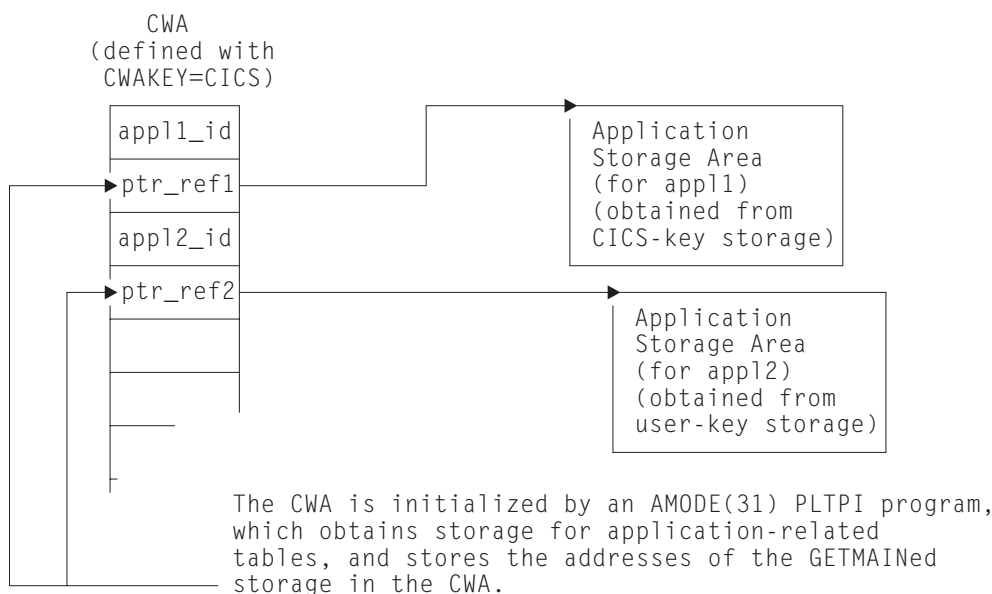


Figure 39. Example of use of CWA in CICS-key storage. This illustrates how the CWA can be used to reference storage that is obtained in user-key or CICS-key storage for use by application programs, while the CWA itself is protected by being in CICS-key storage.

In this illustration, the CWA is not used directly to store application data and constants. The CWA contains pairs of application identifiers and associated addresses, with the address fields containing the addresses of data areas that hold the application-related data. For protection, the CWA is defined with CWAKEY=CICS, therefore the program which in this illustration is a program defined in the program list table post initialization (PLTPI) list, and that loads the CWA with addresses and application identifiers must be defined with EXECKEY(CICS). Any application programs requiring access to the CWA should be

defined with EXECKEY(USER), thereby ensuring the CWA is protected from overwriting by application programs. In Figure 39 on page 184, one of the data areas is obtained from CICS-key storage, while the other is obtained from user-key storage.

In the sample code shown in Figure 40, the program list table post-initialization (PLTPI) program is setting up the application data areas, with pointers to the data stored in the CWA.

This example illustrates how to create global data for use by application programs, with addresses of the data stored in the CWA—for example, by a PLTPI program. The first data area is obtained from CICS-key storage, which is the default on a GETMAIN command issued by a PLTPI program, the second from user-key storage by specifying the USERDATAKEY option. The CWA itself is in CICS-key storage, and PLTPROG is defined with EXECKEY(CICS).

```
ID DIVISION.
PROGRAM-ID. PLTPROG.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 APPLID          PIC X(8)      VALUE SPACES.
77 SYSID           PIC X(4)      VALUE SPACES.
01 COMM-DATA.
   03 AREA-PTR     USAGE IS POINTER.
   03 AREA-LENGTH  PIC S9(8)    COMP.
LINKAGE SECTION.
01 COMMON-WORK-AREA.
   03 APPL-1-ID    PIC X(4).
   03 APPL-1-PTR   USAGE IS POINTER.
   03 APPL-2-ID    PIC X(4).
   03 APPL-2-PTR   USAGE IS POINTER.
PROCEDURE DIVISION.
MAIN-PROCESSING SECTION.
* Obtain APPLID and SYSID values
  EXEC CICS ASSIGN APPLID(APPLID)
                  SYSID(SYSID)

  END-EXEC.
* Set up addressability to the CWA
  EXEC CICS ADDRESS
        CWA(ADDRESS OF COMMON-WORK-AREA)

  END-EXEC.
* Get 12KB of CICS-key storage for the first application ('APP1')
  MOVE 12288      TO AREA-LENGTH.
  EXEC CICS GETMAIN SET(AREA-PTR)
        FLENGTH(AREA-LENGTH)
        SHARED

  END-EXEC.
* Initialize CWA fields and link to load program
* for storage area 1.
  MOVE 'APP1'     TO APPL-1-ID.
  SET APPL-1-PTR  TO AREA-PTR.
  EXEC CICS LINK PROGRAM('LOADTAB1')
        COMMAREA(COMM-DATA)

  END-EXEC.
```

Figure 40. Sample code for loading the CWA (Part 1 of 2)

```

* Get 2KB of user-key storage for the second application ('APP2')
  MOVE 2048      TO  AREA-LENGTH.
  EXEC CICS GETMAIN SET(AREA-PTR)
              FLENGTH(AREA-LENGTH)
              SHARED
              USERDATAKEY
  END-EXEC.
* Initialize CWA fields and link to load program
* for storage area 2.
  MOVE 'APP2'    TO  APPL-2-ID.
  SET APPL-2-PTR TO  AREA-PTR.
  EXEC CICS LINK PROGRAM('LOADTAB2')
              COMMAREA(COMM-DATA)
  END-EXEC.
  EXEC CICS RETURN
  END-EXEC.
MAIN-PROCESSING-EXIT.
GOBACK.

```

Figure 40. Sample code for loading the CWA (Part 2 of 2)

Using the TCTTE user area (TCTUA)

The TCT user area (TCTUA) is an optional extension to the terminal control table entry (TCTTE). Each entry in the TCT specifies whether this extension is present and, if so, how long it is (by means of the `USERAREALEN` attribute of the `TYPETERM` resource definition used for the terminal). See the *CICS Resource Definition Guide* for more information about the `TYPETERM` resource definition.

The system initialization parameters `TCTUALOC` and `TCTUAKEY` specify the location and storage key for all TCTUAs.

- `TCTUALOC=BELOW` or `ANY` specifies whether you want 24- or 31-bit addressability to the TCTUAs, and whether TCTCUAs must be stored below the 16MB line or may be either above or below the line.
- `TCTUAKEY=USER` or `CICS` specifies whether you want the TCTUAs allocated from user-key or CICS-key storage.

TCTUAs have the following characteristics in common with the CWA:

- Minimal processor overhead (only one `ADDRESS` command needed)
- No recovery
- No resource security
- No regulation of use by CICS
- Fixed length
- Unsuitability for large-volume or short-lived data

Unlike the CWA, however, the TCTUA for a particular terminal is usually shared only among transactions using that terminal. It is therefore useful for storing small amounts of data of fairly standard length between a series of transactions in a pseudoconversational sequence. Another difference is that it is not necessarily permanently allocated, because the TCTUA only exists while the TCTTE is set up. For non-autoinstall terminals the TCTUA is allocated from system startup; for autoinstall terminals the TCTUA is allocated when the TCTTE is generated.

Using the TCTUA in this way does not require special discipline among using transactions, because data is always read by the transaction following the one that wrote it. However, if you use TCTUAs to store longer-term data (for example, terminal or operator information needed by an entire application), they require the same care as the CWA to ensure that data used in one transaction does not

overlay data used in another. You should not exceed the length of the allocated TCTUA, because this produces a storage violation.

Using the COMMAREA in RETURN commands

The COMMAREA option of the RETURN command is designed specifically for passing data between successive transactions in a pseudoconversational sequence. It is implemented as a special form of user storage, although the EXEC interface, rather than the application program, issues the GETMAIN and FREEMAIN requests.

The COMMAREA is allocated from the CICS shared subpool in main storage, and is addressed by the TCTTE, between tasks of a pseudoconversational application. The COMMAREA is freed unless it is passed to the next task.

The first program in the next task has automatic addressability to the passed COMMAREA, as if the program had been invoked by either a LINK command or an XCTL command (see “COMMAREA in LINK and XCTL commands” on page 157). You can also use the COMMAREA option of the ADDRESS command to obtain the address of the COMMAREA.

For a COMMAREA passed between successive transactions in a pseudoconversational sequence in a distributed environment, VTAM imposes a limit of 32KB on the size of the total data length. This limit applies to the entire transmitted package, which includes control data added by VTAM. The amount of control data increases if the transmission uses intermediate links.

To summarize:

- Processor overhead is low (equivalent to using COMMAREA with an XCTL command and approximately equal to using main temporary storage).
- It is not recoverable.
- There is no resource security.
- It is not suitable for very large amounts of data (because main storage is used, and it is held until the terminal user responds).
- As with using COMMAREA to transfer data between programs, it is available only to the first program in a transaction, unless that program explicitly passes the data or its address to succeeding programs.

Using a channel on RETURN commands

Instead of using a communication area (COMMAREA), a more modern method of passing data to the next program in a pseudoconversation is to use a **channel**. Channels have several advantages over COMMAREAs—see Benefits of channels. To pass a channel on a RETURN command, you use the CHANNEL option in place of the COMMAREA option.

Channels are described in Enhanced inter-program data transfer: channels as modern-day COMMAREAs.

Using the display screen to share data

You can also store data between pseudoconversational transactions from a 3270 display terminal on the display screen itself. For example, if users make errors in data that they are asked to enter on a screen, the transaction processing the input usually points out the errors on the screen (with highlights or messages), sets the next transaction identifier to point to itself (so that it processes the corrected input), and returns to CICS.

The transaction has two ways of using the **valid** data. It can save it (for example, in COMMAREA), and pass it on for the next time it is run. In this case, the transaction must merge the changed data on the screen with the data from previous entries. Alternatively, it can save the data on the screen by not turning off the modified data tags of the keyed fields.

Saving the data on the screen is very easy to code, but it is not very secure. You are **not** recommended to save screens that contain large amounts of data as errors may occur because of the additional network traffic needed to resend the unchanged data. (This restriction does not apply to locally-attached terminals.)

Secondly, if the user presses the CLEAR key, the screen data is lost, and the transaction must be able to recover from this. You can avoid this by defining the CLEAR key to mean CANCEL or QUIT, if this is appropriate for the application concerned.

Data other than keyed data may also be stored on the screen. This data can be protected from changes (except those caused by CLEAR) and can be nondisplay, if necessary.

Chapter 14. Enhanced inter-program data transfer: channels as modern-day COMMAREAs

Traditionally, CICS programs have used communication areas (COMMAREAs) to exchange data. This topic describes an improved method of transferring data between programs, in amounts that far exceed the 32KB limit that applies to COMMAREAs.

This section contains:

- “Channels: quick start”
- “Using channels: some typical scenarios” on page 192
- “Creating a channel” on page 195
- “The current channel” on page 196
- “The scope of a channel” on page 200
- “Discovering which containers a program's been passed” on page 204
- “Discovering which containers were returned from a link” on page 204
- “CICS read only containers” on page 204
- “Designing a channel: best practices” on page 205
- “Constructing and using a channel: an example” on page 206
- “Channels and BTS activities” on page 207
- “Using channels from JCICS” on page 209
- “Dynamic routing with channels” on page 209
- “Data conversion” on page 210
- “Benefits of channels” on page 215
- “Migrating from COMMAREAs to channels” on page 216

Channels: quick start

Containers and channels

Containers are named blocks of data designed for passing information between programs. You can think of them as “named communication areas (COMMAREAs)”. Programs can pass any number of containers between each other. Containers are grouped together in sets called **channels**. A channel is analogous to a parameter list.

To create named containers and assign them to a channel, a program uses EXEC CICS PUT CONTAINER(*container-name*) CHANNEL(*channel-name*) commands. It can then pass the channel (and its containers) to a second program using the CHANNEL(*channel-name*) option of the EXEC CICS LINK, XCTL, START, or RETURN commands.

The second program can read containers passed to it using the EXEC CICS GET CONTAINER(*container-name*) command. This command reads the named container belonging to the channel that the program was invoked with.

If the second program is invoked by a LINK command, it can also return containers to the calling program. It can do this by creating new containers, or by reusing existing containers.

Channels and containers are visible only to the program that creates them and the programs they are passed to. When these programs terminate, CICS automatically destroys the containers and their storage.

Channel containers are not recoverable. If you need to use recoverable containers, use CICS business transaction services (BTS) containers. The relationship between channel and BTS containers is described in “Channels and BTS activities” on page 207.

Basic examples

Figure 41 on page 191 shows a COBOL program, CLIENT1, that:

1. Uses PUT CONTAINER(*container-name*) CHANNEL(*channel-name*) commands to create a channel called inqcustrec and add two containers, custno and branchno, to it; these contain a customer number and a branch number, respectively.
2. Uses a LINK PROGRAM(*program-name*) CHANNEL(*channel-name*) command to link to program SERVER1, passing the inqcustrec channel.
3. Uses a GET CONTAINER(*container-name*) CHANNEL(*channel-name*) command to retrieve the customer record returned by SERVER1. The customer record is in the custrec container of the inqcustrec channel.

Note that the same COBOL copybook, INQINTC, is used by both the client and server programs. Line 3 and lines 5 through 7 of the copybook represent the INQUIRY-CHANNEL and its containers. These lines are not strictly necessary to the working of the programs, because containers and channels are created simply by being *named* (on, for example, PUT CONTAINER commands); they do not have to be defined. However, the inclusion of these lines in the copybook used by both programs makes for easier maintenance; they record the names of the containers used.

Recommendation

For ease of maintenance of a client/server application that uses a channel, create a copybook that records the names of the containers used and defines the data fields that map to the containers. Include the copybook in both the client and the server program.

Note: This example shows two COBOL programs. The same techniques can be used in any of the other languages supported by CICS. However, *for COBOL programs only*, if the server program uses the SET option (instead of INTO) on the EXEC CICS GET CONTAINER command, the structure of the storage pointed to by SET must be defined in the LINKAGE section of the program. This means that you will require two copybooks rather than one. The first, in the WORKING-STORAGE section of the program, names the channel and containers used. The second, in the LINKAGE section, defines the storage structure.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. CLIENT1.

WORKING-STORAGE SECTION.

    COPY INQINTC
    *               copybook INQINTC
    * Channel name
    * 01 INQUIRY-CHANNEL PIC X(16) VALUE 'inqcustrec'.
    * Container names
    * 01 CUSTOMER-NO      PIC X(16) VALUE 'custno'.
    * 01 BRANCH-NO        PIC X(16) VALUE 'branchno'.
    * 01 CUSTOMER-RECORD  PIC X(16) VALUE 'custrec'.
    * Define the data fields used by the program
    * 01 CUSTNO           PIC X(8).
    * 01 BRANCHNO        PIC X(5).
    * 01 CREC.
    * 02 CUSTNAME         PIC X(80).
    * 02 CUSTADDR1        PIC X(80).
    * 02 CUSTADDR2        PIC X(80).
    * 02 CUSTADDR3        PIC X(80).

PROCEDURE DIVISION.
MAIN-PROCESSING SECTION.

*
* INITIALISE CUSTOMER RECORD
*
*     ... CREATE CUSTNO and BRANCHNO
*
* GET CUSTOMER RECORD
*
*     EXEC CICS PUT CONTAINER(CUSTOMER-NO) CHANNEL(INQUIRY-CHANNEL)
*                   FROM(CUSTNO) FLENGTH(LENGTH OF CUSTNO)
*                   END-EXEC
*     EXEC CICS PUT CONTAINER(BRANCH-NO) CHANNEL(INQUIRY-CHANNEL)
*                   FROM(BRANCHNO) FLENGTH(LENGTH OF BRANCHNO)
*                   END-EXEC
*
*     EXEC CICS LINK PROGRAM('SERVER1') CHANNEL(INQUIRY-CHANNEL) END-EXEC
*
*     EXEC CICS GET CONTAINER(CUSTOMER-RECORD) CHANNEL(INQUIRY-CHANNEL)
*                   INTO(CREC) END-EXEC
*
*
* PROCESS CUSTOMER RECORD
*
*     ... FURTHER PROCESSING USING CUSTNAME and CUSTADDR1 etc...
*
*     EXEC CICS RETURN END-EXEC
*
* EXIT.

```

Figure 41. A simple example of a program that creates a channel and passes it to a second program

Figure 42 on page 192 shows the SERVER1 program linked to by CLIENT1. SERVER1 retrieves the data from the custno and branchno containers it has been passed, and uses it to locate the full customer record in its database. It then creates a new container, custrec, on the same channel, and returns the customer record in it.

Note that the programmer hasn't specified the CHANNEL keyword on the GET and PUT commands in SERVER1: if the channel isn't specified explicitly, the current channel is used—that is, the channel that the program was invoked with.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. SERVER1.

WORKING-STORAGE SECTION.

    COPY INQINTC
    *               copybook INQINTC
    * Channel name
    * 01 INQUIRY-CHANNEL PIC X(16) VALUE 'inqcustrec'.
    * Container names
    * 01 CUSTOMER-NO     PIC X(16) VALUE 'custno'.
    * 01 BRANCH-NO       PIC X(16) VALUE 'branchno'.
    * 01 CUSTOMER-RECORD PIC X(16) VALUE 'custrec'.
    * Define the data fields used by the program
    * 01 CUSTNO          PIC X(8).
    * 01 BRANCHNO        PIC X(5).
    * 01 CREC.
    * 02 CUSTNAME         PIC X(80).
    * 02 CUSTADDR1        PIC X(80).
    * 02 CUSTADDR2        PIC X(80).
    * 02 CUSTADDR3        PIC X(80).

PROCEDURE DIVISION.
MAIN-PROCESSING SECTION.

    EXEC CICS GET CONTAINER(CUSTOMER-NO)
                INTO(CUSTNO) END-EXEC
    EXEC CICS GET CONTAINER(BRANCH-NO)
                INTO(BRANCHNO) END-EXEC

    ... USE CUSTNO AND BRANCHNO TO FIND CREC IN A DATABASE

    EXEC CICS PUT CONTAINER(CUSTOMER-RECORD)
                FROM(CREC)
                FLENGTH(LENGTH OF CREC) END-EXEC

    EXEC CICS RETURN END-EXEC

EXIT.

```

Figure 42. A simple example of a linked to program that retrieves data from the channel it has been passed. This program is linked-to by program CLIENT1 shown in Figure 41 on page 191.

Using channels: some typical scenarios

Channels and containers provide a powerful way to pass data between programs. This section contains some examples of how channels can be used.

One channel, one program

Figure 43 on page 193 shows the simplest scenario—a “standalone” program with a single channel with which it can be invoked.

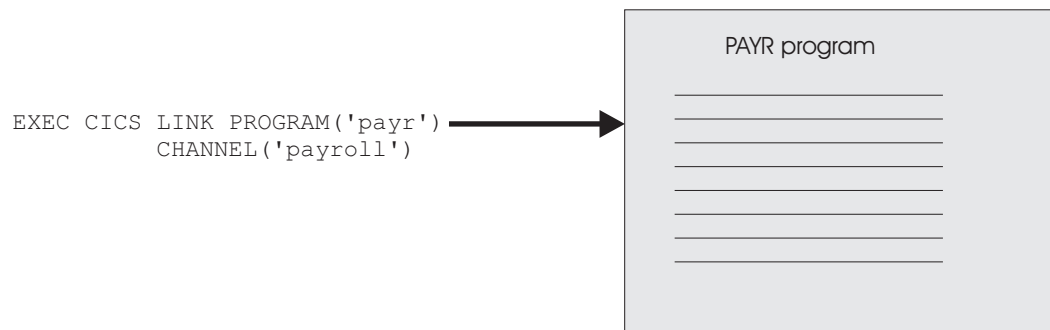


Figure 43. A standalone program with a single channel

One channel, several programs (a component)

In Figure 44, there is a single channel to the top-level program in a set of inter-related programs. The set of programs within the shaded area can be regarded as a “component”. The client program “sees” only the external channel and has no knowledge of the processing that takes place nor of the existence of the back-end programs.

Inside the component, the programs can pass the channel between themselves. Alternatively, a component program could, for example, pass a subset of the original channel, by creating a new channel and adding one or more containers from the original channel.

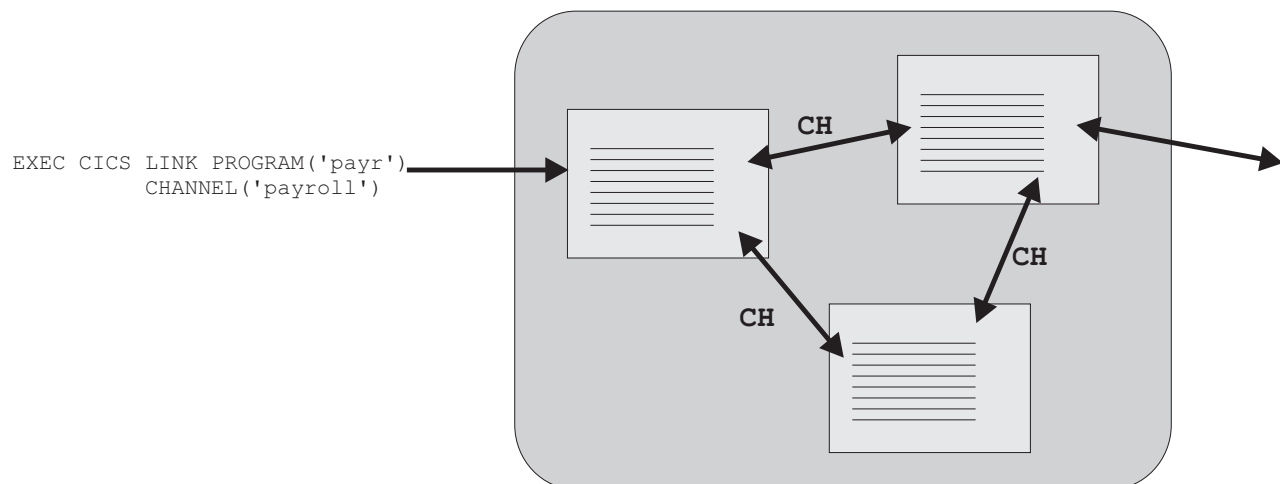


Figure 44. A “component”—a set of related programs invoked through a single external channel. “CH” indicates that the programs within the component can pass channels between themselves.

Several channels, one component

As in the previous example, we have a set of inter-related programs that can be regarded as a component. However, this time there are two, alternative, external channels with which the component can be invoked. The top-level program in the component issues an EXEC CICS ASSIGN CHANNEL command to determine which channel it has been invoked with, and tailors its processing accordingly.

The “loose coupling” between the client program and the component permits easy evolution. That is, the client and the component can be upgraded at different times. For example, first the component could be upgraded to handle a third channel, consisting of a different set of containers from the first or second channels. Next, the client program could be upgraded (or a new client written) to pass the third channel.

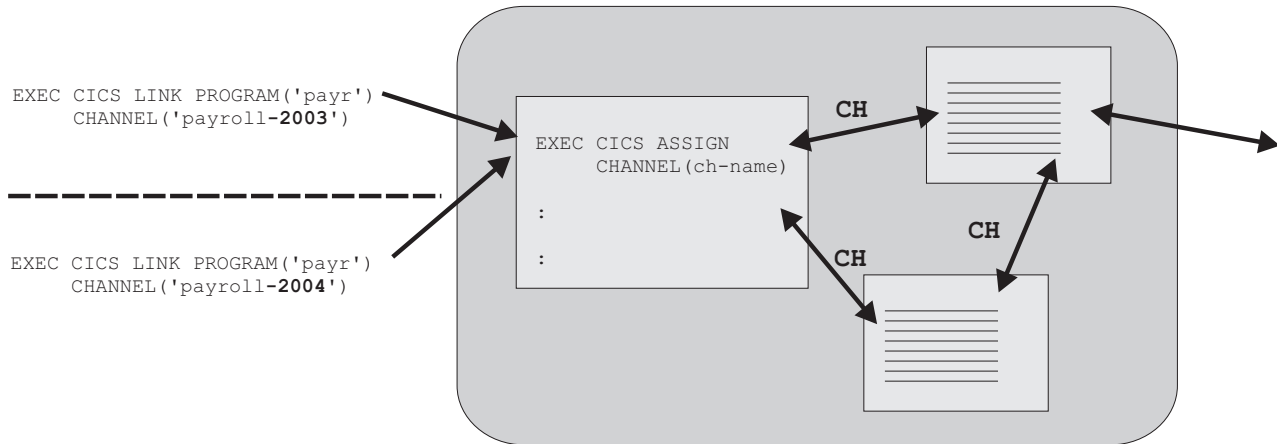


Figure 45. Multiple external channels to the same component. “CH” indicates that the programs within the component may pass channels between themselves.

Multiple interactive components

Figure 46 on page 195 shows a “Human resources” component and a “Payroll” component, each with a channel with which it can be invoked. The Payroll component is invoked from both a standalone program and the Human resources component.

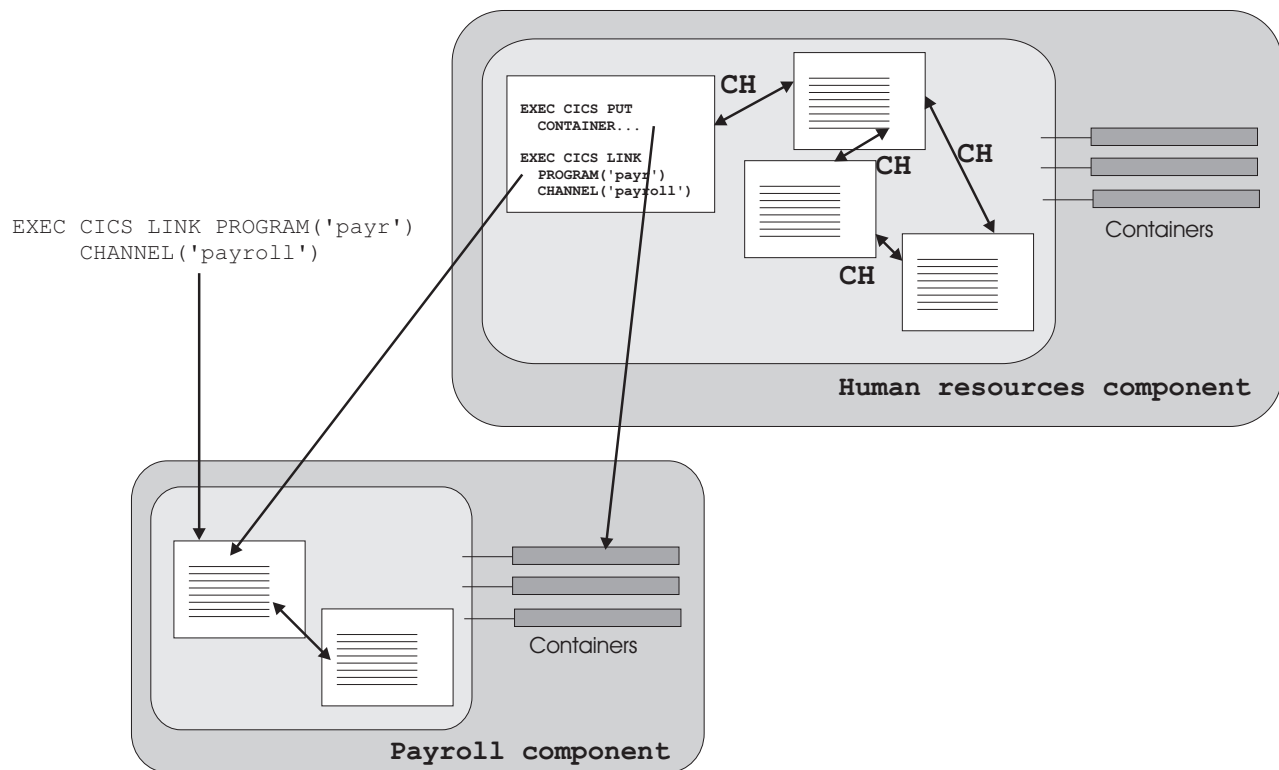


Figure 46. Multiple components which interact through their channels

Creating a channel

About this task

You create a channel by naming it on one of the following commands:

```
EXEC CICS LINK PROGRAM CHANNEL
EXEC CICS MOVE CONTAINER CHANNEL TOCHANNEL
EXEC CICS PUT CONTAINER CHANNEL
EXEC CICS RETURN TRANSID CHANNEL
EXEC CICS START TRANSID CHANNEL
EXEC CICS XCTL PROGRAM CHANNEL
```

If the channel doesn't already exist, within the current program scope, it is created.

The most straightforward way to create a channel, and populate it with containers of data, is to issue a succession of `EXEC CICS PUT CONTAINER(container-name) CHANNEL(channel-name) FROM(data_area)` commands. The first PUT command creates the channel (if it doesn't already exist), and adds a container to it; the subsequent commands add further containers to the channel. If the containers already exist, their contents are overwritten by the new data.

An alternative way to add containers to a channel is to move them from another channel. To do this, use the following command:

```
EXEC CICS MOVE CONTAINER(container-name) AS(container-new-name)
CHANNEL(channel-name1) TOCHANNEL(channel-name2)
```

Note:

1. If the CHANNEL or TOCHANNEL option isn't specified, the current channel is implied.
2. The source channel must be in program scope.
3. If the target channel doesn't already exist, within the current program scope, it is created.
4. If the source container doesn't exist, an error occurs.
5. If the target container doesn't already exist, it is created; if it already exists, its contents are overwritten.

You can use MOVE CONTAINER, instead of GET CONTAINER and PUT CONTAINER, as a more efficient way of transferring data between channels.

If the channel named on the following commands doesn't already exist, within the current program scope, an *empty* channel is created:

- EXEC CICS LINK PROGRAM CHANNEL(*channel-name*)
- EXEC CICS RETURN TRANSID CHANNEL(*channel-name*)
- EXEC CICS START TRANSID CHANNEL(*channel-name*)
- EXEC CICS XCTL PROGRAM CHANNEL(*channel-name*)

The current channel

A program's **current channel** is the channel (if any) with which it was invoked. The program can create other channels. However, the *current* channel, for a particular invocation of a particular program, does not change. It is analogous to a parameter list.

Current channel example, with LINK commands

The following figure illustrates the origin of a program's current channel. It shows five interactive programs. Program A is a top-level program started by, for example, a terminal end-user. It isn't started by a program and doesn't have a current channel.

B, C, D, and E are second-, third-, fourth-, and fifth-level programs, respectively.

Program B's current channel is X, passed by the CHANNEL option on the EXEC CICS LINK command issued by program A. Program B modifies channel X by adding one container and deleting another.

Program C's current channel is also X, passed by the CHANNEL option on the EXEC CICS LINK command issued by program B.

Program D has no current channel, because C doesn't pass it one.

Program E's current channel is Y, passed by the CHANNEL option on the EXEC CICS LINK command issued by D.

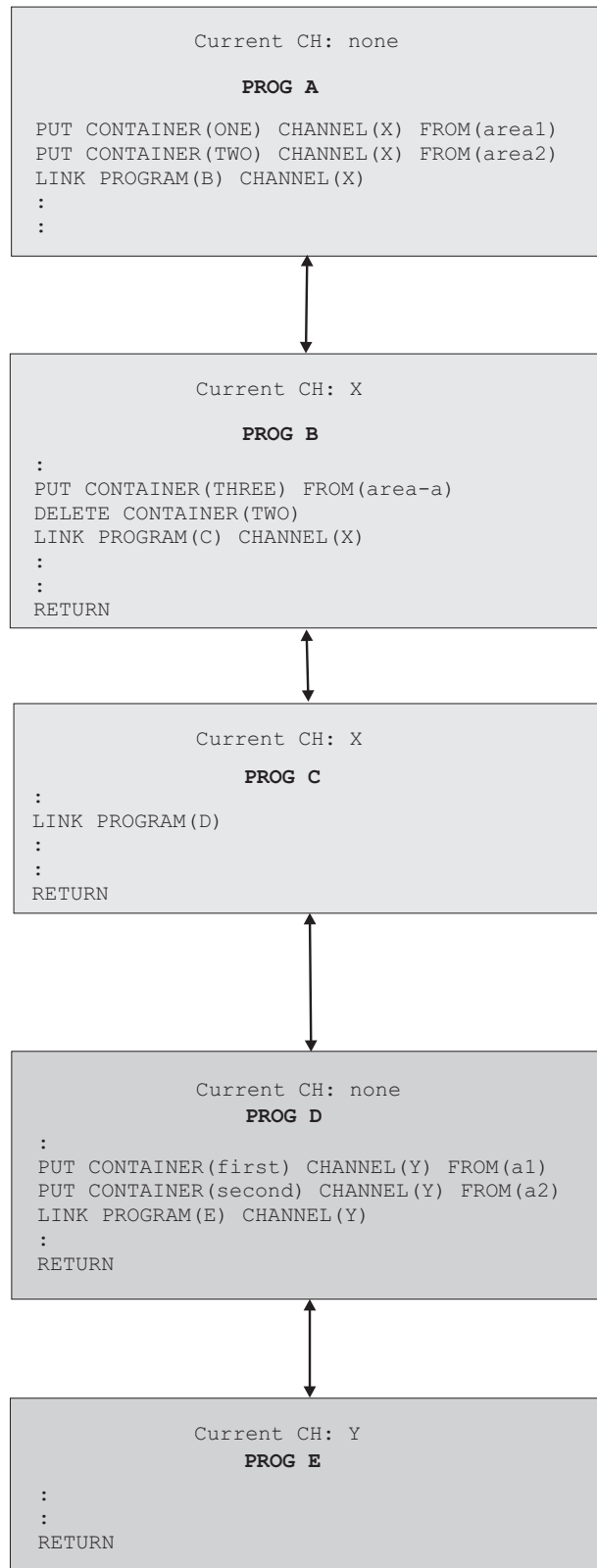


Figure 47. Current channel: example with LINK commands

The following table lists the name of the current channel (if any) of each of the five programs shown in the previous figure.

Table 9. The current channels of interactive programs—example with LINK commands

Prog.	Current CH	Issues commands	Comments
A	None	<pre> . EXEC CICS PUT CONTAINER(ONE) CHANNEL(X) FROM(area1) EXEC CICS PUT CONTAINER(TWO) CHANNEL(X) FROM(area2) EXEC CICS LINK PROGRAM(B) CHANNEL(X) . </pre>	<p>Program A creates channel X and passes it to program B.</p> <p>Note that, by the time control is returned to program A by program B, the X channel has been modified—it doesn't contain the same set of containers as when it was created by program A. (Container TWO has been deleted and container THREE added by program B.)</p>
B	X	<pre> . EXEC CICS PUT CONTAINER(THREE) FROM(area-a) EXEC CICS DELETE CONTAINER(TWO) EXEC CICS LINK PROGRAM(C) CHANNEL(X) . . EXEC CICS RETURN </pre>	<p>Program B modifies channel X (its current channel) by adding and deleting containers, and passes the modified channel to program C.</p> <p>Program B doesn't need to specify the CHANNEL option on the PUT CONTAINER and DELETE CONTAINER commands; its current channel is implied.</p>
C	X	<pre> . EXEC CICS LINK PROGRAM(D) . . EXEC CICS RETURN </pre>	<p>Program C links to program D, but does not pass it a channel.</p>
D	None	<pre> . EXEC CICS PUT CONTAINER(first) CHANNEL(Y) FROM(a1) EXEC CICS PUT CONTAINER(second) CHANNEL(Y) FROM(a2) EXEC CICS LINK PROGRAM(E) CHANNEL(Y) . . EXEC CICS RETURN </pre>	<p>Program D creates a new channel, Y, which it passes to program E.</p>
E	Y	<pre> . RETURN . </pre>	<p>Program E performs some processing on the data it's been passed and returns.</p>

Current channel example, with XCTL commands

Figure 48 on page 199 shows four interactive programs. A1 is a top-level program started by, for example, a terminal end-user. It isn't started by a program and doesn't have a current channel. B1, B2, and B3 are all second-level programs.

B1's current channel is X, passed by the CHANNEL option on the EXEC CICS LINK command issued by A1.

B2 has no current channel, because B1 doesn't pass it one.

B3's current channel is Y, passed by the CHANNEL option on the EXEC CICS XCTL command issued by B2.

When B3 returns, channel Y and its containers are deleted by CICS.

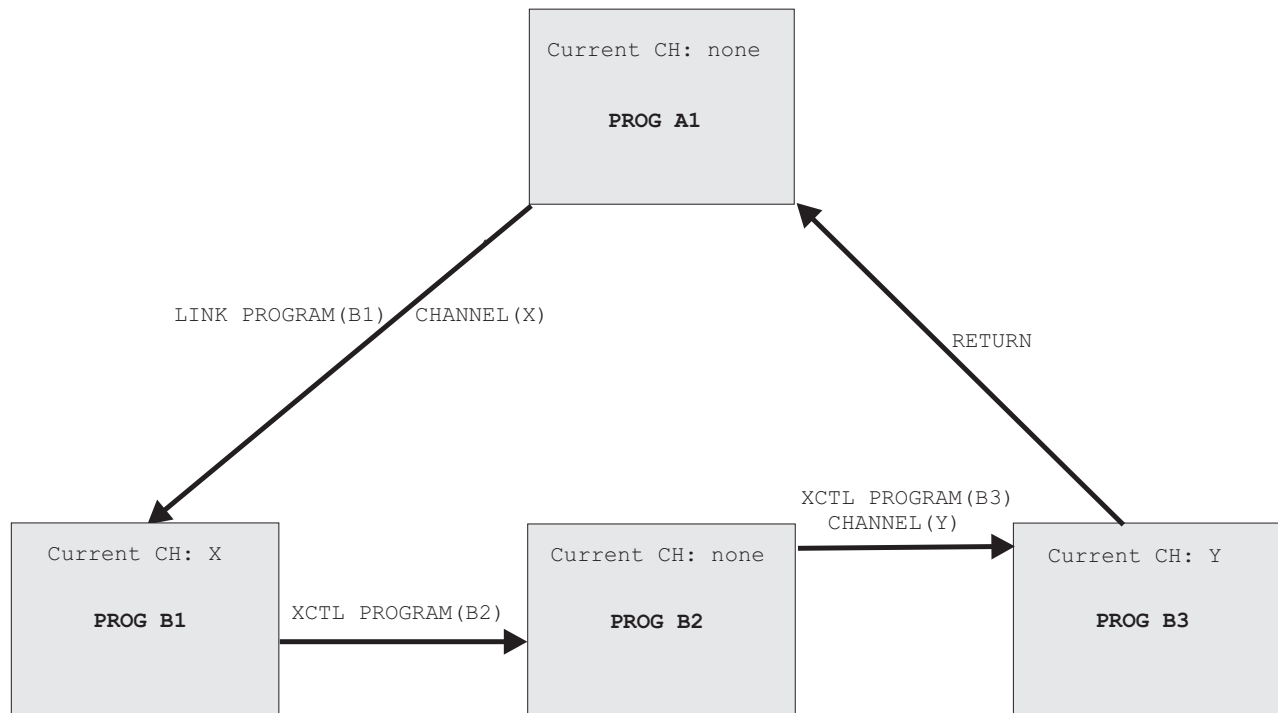


Figure 48. Current channels—example, with XCTL commands

The following table lists the name of the current channel (if any) of each of the four programs shown in Figure 48.

Table 10. The current channels of interactive programs—example

Program	Current channel	Issues command
A1	None	. EXEC CICS LINK PROGRAM(B1) CHANNEL(X) .
B1	X	. EXEC CICS XCTL PROGRAM(B2) .
B2	None	. EXEC CICS XCTL PROGRAM(B3) CHANNEL(Y) .
B3	Y	. EXEC CICS RETURN .

Current channel: START and RETURN commands

Besides EXEC CICS LINK and XCTL, two other commands can be used to invoke a program and pass it a channel:

EXEC CICS START TRANSID(*transid*) CHANNEL(*channel-name*)

The program that implements the started transaction (or the first program, if there are more than one) is passed the channel, which becomes its current channel.

EXEC CICS RETURN TRANSID(*tranid*) CHANNEL(*channel-name*)

The CHANNEL option is valid only:

- On pseudoconversational RETURNS—that is, on RETURN commands that specify, on the TRANSID option, the next transaction to be run at the user terminal.
- If issued by a program at the highest logical level—that is, a program that returns control to CICS.

The program that implements the next transaction is passed the channel, which becomes its current channel.

The scope of a channel

The **scope** of a channel is the code from which it can be accessed.

Scope example, with LINK commands

The following figure shows the same five interactive programs previously described in “Current channel example, with LINK commands”.

The scope of the X channel—the code from which it can be accessed—is programs A, B, and C.

The scope of the Y channel is programs D and E.

Note that, by the time control is returned to program A by program B, the X channel has been modified—it doesn't contain the same set of containers as when it was created by program A.

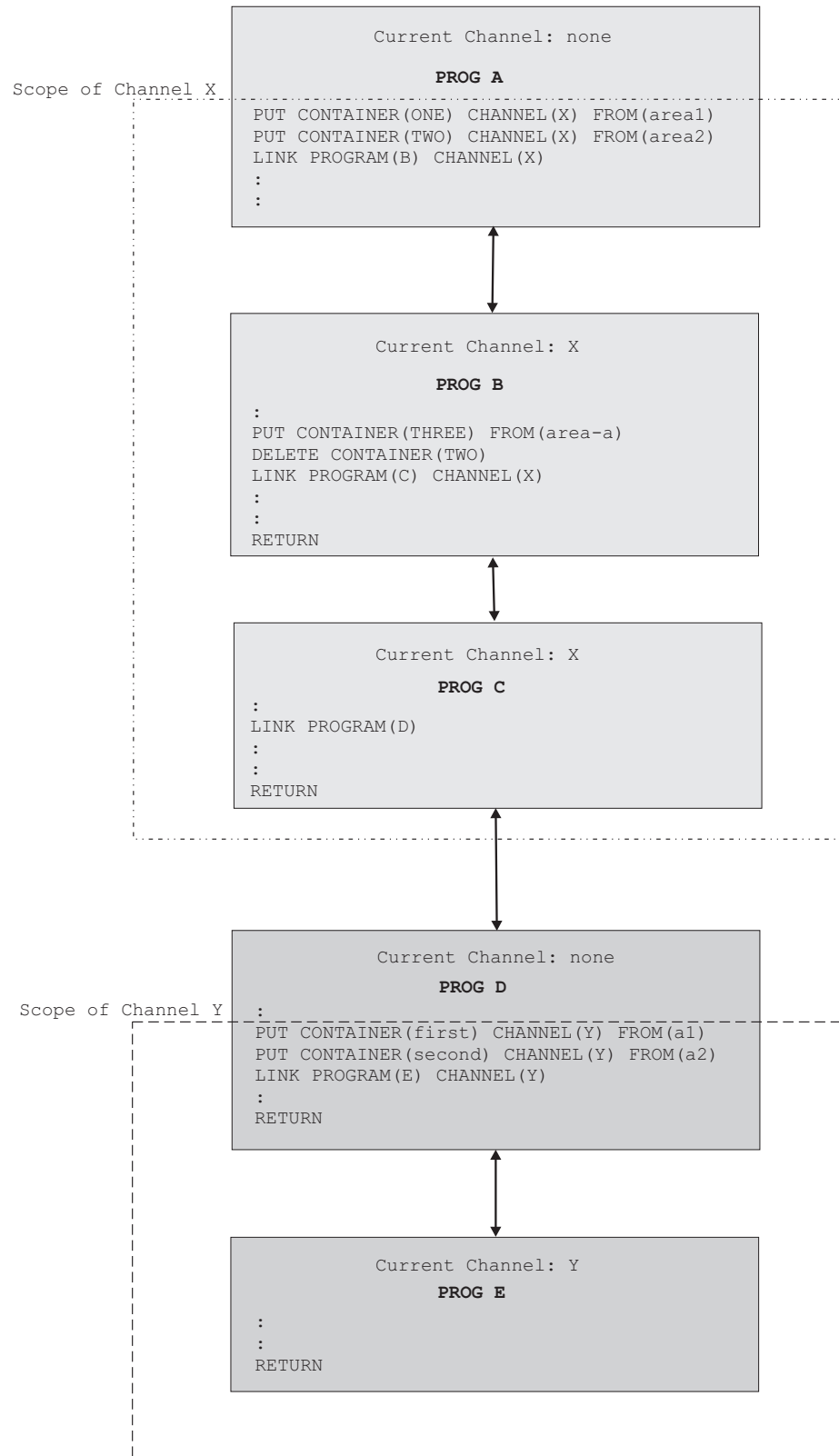


Figure 49. The scope of a channel—example showing LINK commands

The following table lists the name and scope of the current channel (if any) of each of the five programs shown in the previous figure.

Table 11. The scope of a channel—example with LINK commands

Program	Current channel	Scope of channel
A	None	Not applicable
B	X	A, B, C
C	X	A, B, C
D	None	Not applicable
E	Y	D, E

Scope example, with LINK and XCTL commands

Figure 50 on page 203 shows the same four interactive programs previously described in “Current channel example, with XCTL commands”, plus a third-level program, C1, that is invoked by an EXEC CICS LINK command from program B1.

The scope of the X channel is restricted to A1 and B1.

The scope of the Y channel is B2 and B3.

The scope of the Z channel is B1 and C1.

Note that, by the time control is returned to program A1 by program B3, it's possible that the X channel may have been modified by program B1—it might not contain the same set of containers as when it was created by A1.

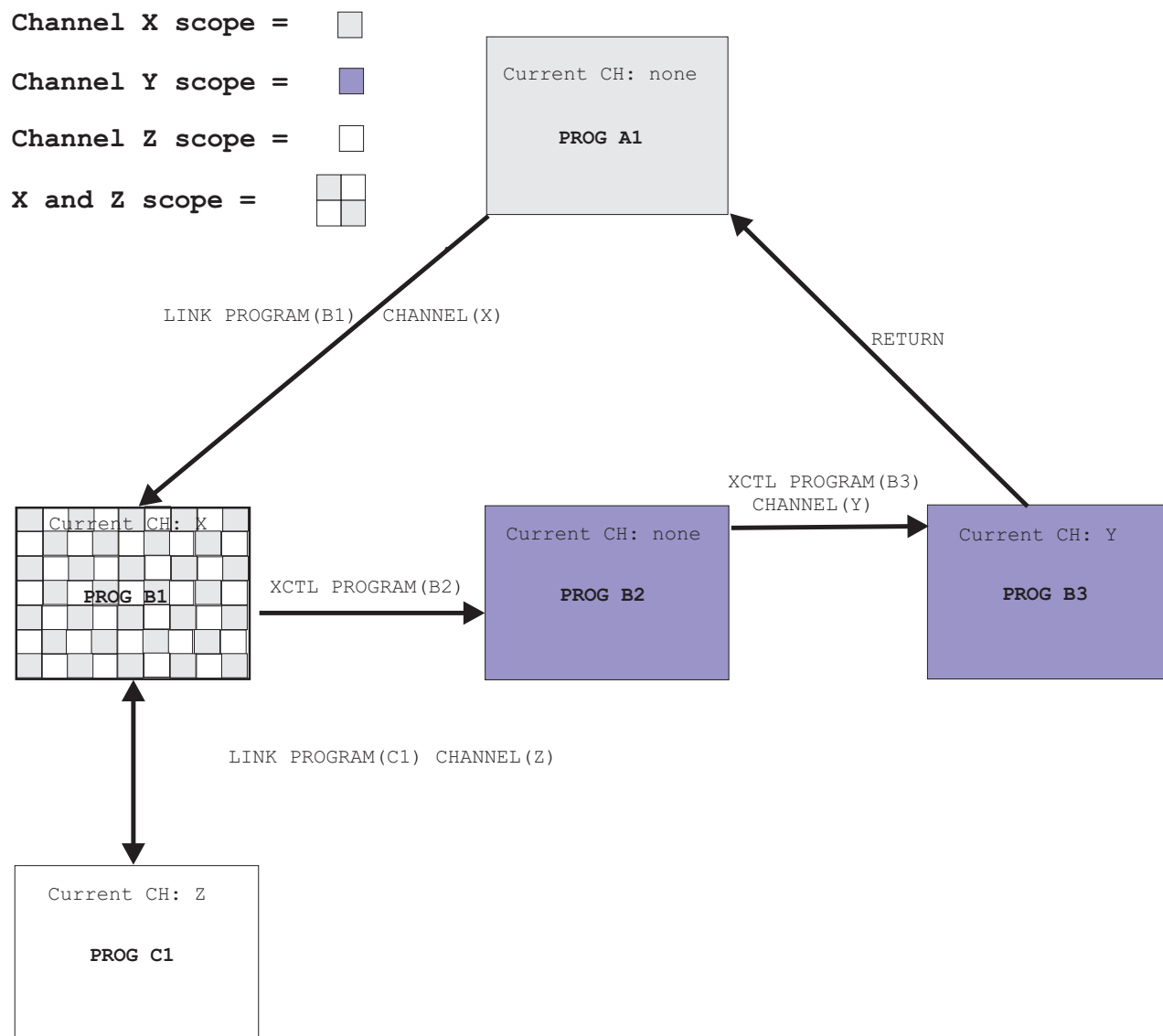


Figure 50. The scope of a channel—example showing LINK and XCTL commands

The following table lists the name and scope of the current channel (if any) of each of the five programs shown in Figure 50.

Table 12. The scope of a channel—example with LINK and XCTL commands

Program	Current channel	Scope of channel
A1	None	Not applicable
B1	X	A1 and B1
B2	None	Not applicable
B3	Y	B2 and B3
C1	Z	B1 and C1

Discovering which containers a program's been passed

Typically, programs that exchange a channel are written to handle that channel. That is, both client and server programs know the name of the channel, and the names and number of the containers in the channel. However, if, for example, a server program or component is written to handle more than one channel, on invocation it must discover which of the possible channels it's been passed.

A program can discover its current channel—that is, the channel with which it was invoked—by issuing an EXEC CICS ASSIGN CHANNEL command. (If there is no current channel, the command returns blanks.)

The program can also (should it need to) get the names of the containers in its current channel by browsing. Typically, this is not necessary. A program written to handle several channels is often coded to be aware of the names and number of the containers in each possible channel.

To get the names of the containers in the current channel, use the browse commands:

- EXEC CICS STARTBROWSE CONTAINER BROWSETOKEN(*data-area*) .
- EXEC CICS GETNEXT CONTAINER(*data-area*) BROWSETOKEN(*token*).
- EXEC CICS ENDBROWSE CONTAINER BROWSETOKEN(*token*).

Having retrieved the name of its current channel and, if necessary, the names of the containers in the channel, a server program can adjust its processing to suit the kind of data that it's been passed.

Discovering which containers were returned from a link

A program creates a channel, which it passes to a second program by means of an EXEC CICS LINK PROGRAM(*program-name*) CHANNEL(*channel-name*) command. The second program performs some processing on the data it's been passed, and returns the results in the same channel (its current channel).

On return, the first program knows the name of the channel which has been returned, but not necessarily the names of the containers in the channel. (Does the returned channel contain the same containers as the passed channel, or has the second program deleted some or created others?) The first program can discover the container-names by browsing. To do this, it uses the commands:

- EXEC CICS STARTBROWSE CONTAINER BROWSETOKEN(*data-area*)
CHANNEL(*channel-name*).
- EXEC CICS GETNEXT CONTAINER(*data-area*) BROWSETOKEN(*token*).
- EXEC CICS ENDBROWSE CONTAINER BROWSETOKEN(*token*).

CICS read only containers

CICS can create channels and containers for its own use, and pass them to user programs. In some cases CICS marks these containers as read only, so that the user program cannot modify data which CICS needs on return from the user program.

User programs cannot create read only containers.

You cannot overwrite, move, or delete a read only container. Thus, if you specify a read only container on a PUT CONTAINER, MOVE CONTAINER, or DELETE CONTAINER command an INVREQ condition occurs.

Designing a channel: best practices

About this task

It's possible to use containers to pass data in the same way as communication areas (COMMAREAs) have traditionally been used. However, channels have several advantages over COMMAREAs (see "Benefits of channels" on page 215) and it pays to design your channels to make the most of these improvements.

At the end of a DPL call, input containers that have not been changed by the server program are not returned to the client. Input containers whose contents have been changed by the server program, and containers created by the server program, are returned. Therefore, for optimal DPL performance:

- Use separate containers for input and output data.
- The server program, not the client, should create the output containers.
- Use separate containers for read-only and read-write data.
- If a structure is optional, make it a separate container.
- Use dedicated containers for error information.

Here are some general tips on designing a channel. They include and expand on the recommendations for achieving optimal DPL performance.

- Use separate containers for input and output data. This leads to:
 - Better encapsulation of the data, making your programs easier to maintain.
 - Greater efficiency when a channel is passed on a DPL call, because smaller containers flow in each direction.
- The server program, not the client, should create the output containers. If the client creates them, empty containers will be sent to the server region.
- Use separate containers for read-only and read-write data. This leads to:
 - A simplification of your copybook structure, making your programs easier to understand.
 - Avoidance of the problems with REORDER overlays.
 - Greater transmission efficiency between CICS regions, because read-only containers sent to a server region will not be returned.
- Use separate containers for each structure. This leads to:
 - Better encapsulation of the data, making your programs easier to understand and maintain.
 - Greater ease in changing one of the structures, because you don't need to recompile the entire component.
 - The ability to pass a subset of the channel to sub-components, by using the MOVE CONTAINER command to move containers between channels.
- If a structure is optional, make it a separate container. This leads to greater efficiency, because the structure is passed only if the container is present.
- Use dedicated containers for error information. This leads to:
 - Better documentation of what is error information.
 - Greater efficiency, because:
 1. The structure containing the error information is passed back only if an error occurs.
 2. It is more efficient to check for the presence of an error container by issuing a GET CONTAINER(*known-error-container-name*) command (and possibly receiving a NOTFOUND condition) than it is to initiate a browse of the containers in the channel.

```
#
#
```

- When you need to pass data of different types—for example, character data in codepage1 and character data in codepage2—use separate containers for each type, rather than one container with a complicated structure. This improves your ability to move between different code pages.
- When you need to pass a large amount of data, split it between multiple containers, rather than put it all into one container.

When a channel is passed to a remote program or transaction, passing a large amount of data may affect performance. This is particularly true if the local and remote regions are connected by an ISC, rather than MRO, connection.

CAUTION:

```
#
#
```

Take care not to create so many large containers that you limit the amount of storage available to other applications.

For information about migrating programs that use COMMAREAs to use channels instead, see “Migration to the new function” on page 216.

Constructing and using a channel: an example

Figure 51 shows a CICS client program that:

1. Uses EXEC CICS PUT CONTAINER commands to construct (and put data in) a set of containers. The containers are all part of the same named channel—“payroll-2004”.
2. Issues an EXEC CICS LINK command to invoke the PAYR server program, passing it the payroll-2004 channel.
3. Issues an EXEC CICS GET CONTAINER command to retrieve the server's program output, which it knows will be in the status container of the payroll-2004 channel.

```
* create the employee container on the payroll-2004 channel
EXEC CICS PUT CONTAINER('employee') CHANNEL('payroll-2004') FROM('John Doe')

* create the wage container on the payroll-2004 channel
EXEC CICS PUT CONTAINER('wage') CHANNEL('payroll-2004') FROM('100')

* invoke the payroll service, passing the payroll-2004 channel
EXEC CICS LINK PROGRAM('PAYR') CHANNEL('payroll-2004')

* examine the status returned on the payroll-2004 channel
EXEC CICS GET CONTAINER('status') CHANNEL('payroll-2004') INTO(stat)
```

Figure 51. How a client program can construct a channel, pass it to a server program, and retrieve the server's output

Figure 52 on page 207 shows part of the PAYR server program invoked by the client. The server program:

1. Queries the channel with which it's been invoked.
2. Issues EXEC CICS GET CONTAINER commands to retrieve the input from the employee and wage containers of the payroll-2004 channel.
3. Processes the input data.
4. Issues an EXEC CICS PUT CONTAINER command to return its output in the status container of the payroll-2004 channel.

```

"PAYR", CICS COBOL server program

* discover which channel I've been invoked with
EXEC CICS ASSIGN CHANNEL(ch_name)
:
WHEN ch_name 'payroll-2004'
  * my current channel is "payroll-2004"
  * get the employee passed into this program
  EXEC CICS GET CONTAINER('employee') INTO(emp)
  * get the wage for this employee
  EXEC CICS GET CONTAINER('wage') INTO(wge)
  :
  * process the input data
  :
  :
  * return the status to the caller by creating the status container
  * on the payroll channel and putting a value in it
  EXEC CICS PUT CONTAINER('status') FROM('OK')
  :
  :
WHEN ch_name 'payroll-2005'
  * my current channel is "payroll-2005"
  :
  :
  :

```

Figure 52. How a server program can query the channel it's been passed, retrieve data from the channel's containers, and return output to the caller

Channels and BTS activities

The PUT, GET, MOVE, and DELETE CONTAINER commands used to build and interact with a channel are similar to those used in CICS business transaction services (BTS) applications. (For information about BTS, see the *CICS Business Transaction Services* manual.) Thus, programmers with experience of BTS will find it easy to use containers in non-BTS applications. Furthermore, server programs that use containers can be called from both channel and BTS applications. An example of this is shown in Figure 53 on page 208.

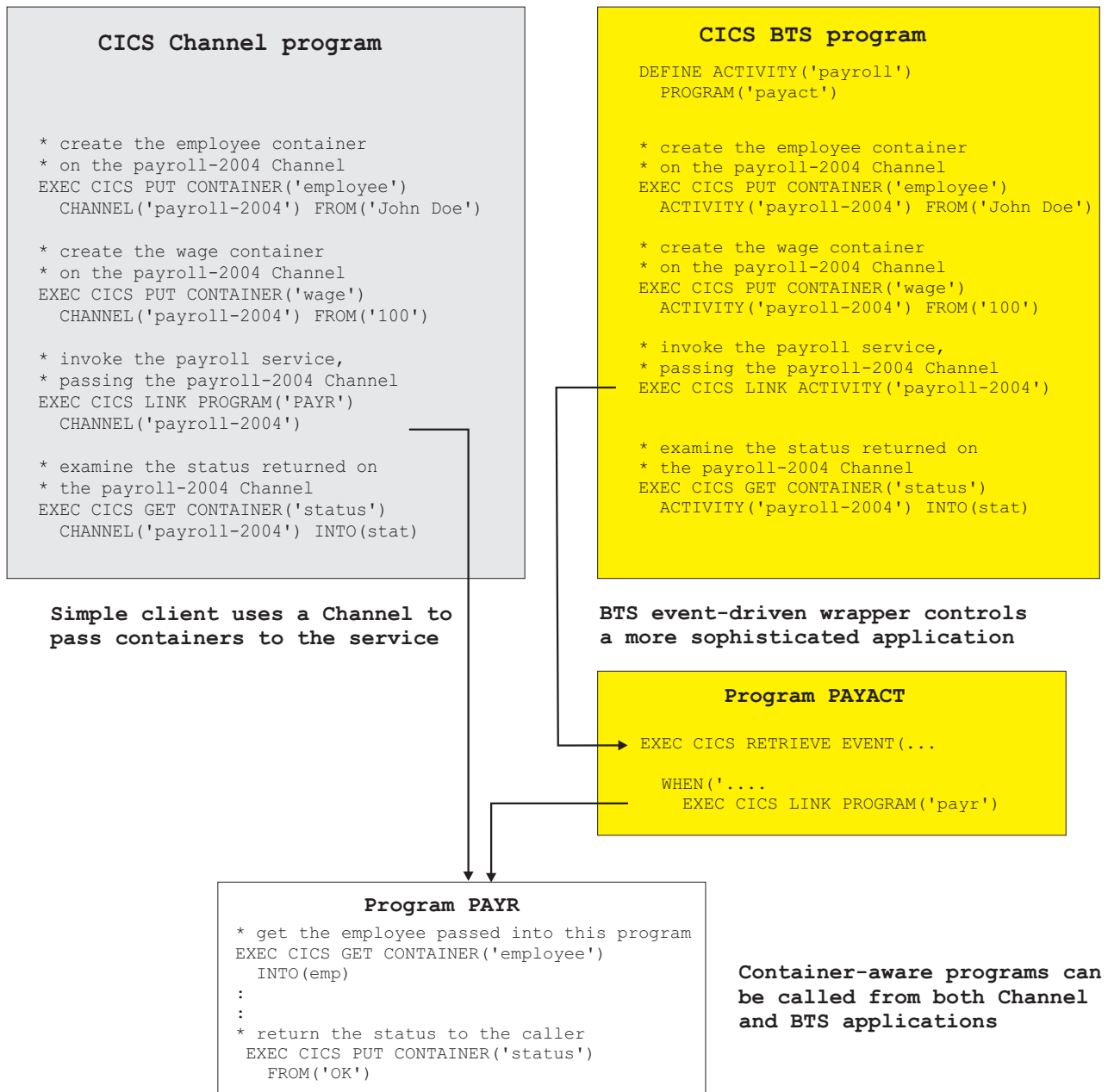


Figure 53. Channels and BTS activities

Context

As shown in Figure 53, a program that issues container commands can be used, without change, as part of a channel application or as part of a BTS activity.

For a program to be used in both a channel and a BTS context, the container commands that it issues must not specify any options that identify them as either channel or BTS commands. The options to be avoided on each of the container commands are:

DELETE CONTAINER
ACQACTIVITY (BTS-specific)

ACQPROCESS (BTS-specific)
ACTIVITY (BTS-specific)
CHANNEL (channel-specific)
PROCESS (BTS-specific)

GET CONTAINER

ACQACTIVITY (BTS-specific)
ACQPROCESS (BTS-specific)
ACTIVITY (BTS-specific)
CHANNEL (channel-specific)
INTOCCSID (channel-specific)
PROCESS (BTS-specific)

MOVE CONTAINER

FROMACTIVITY (BTS-specific)
CHANNEL (channel-specific)
FROMPROCESS (BTS-specific)
TOACTIVITY (BTS-specific)
TOCHANNEL (channel-specific)
TOPROCESS (BTS-specific)

PUT CONTAINER

ACQACTIVITY (BTS-specific)
ACQPROCESS (BTS-specific)
ACTIVITY (BTS-specific)
CHANNEL (channel-specific)
DATATYPE (channel-specific)
FROMCCSID (channel-specific)
PROCESS (BTS-specific)

When a container command is executed, CICS analyzes the context (channel, BTS, or neither) in which it occurs, in order to determine how to process the command. To determine the context, CICS uses the following sequence of tests:

1. Channel: does the program have a current channel?
2. BTS: is the program part of a BTS activity?
3. None: the program has no current channel and is not part of a BTS activity. It therefore has no context in which to execute container commands. The command is rejected with an INVREQ condition and a RESP2 value of 4.

Using channels from JCICS

For information about using channels with JCICS, see *Java Applications in CICS*.

Dynamic routing with channels

EXEC CICS LINK and EXEC CICS START commands, which can pass either COMMAREAs or channels, can be dynamically routed. Thus the following types of channel-related request can be dynamically routed:

- Program-link (DPL) requests
- Transactions started by terminal-related START requests
- Non-terminal-related START requests

The routing program is passed, in the DYRCHANL field of its communication area, the name of the channel, if any, associated with the program-link or START command. The DYRCHANL field applies only to the three types of request listed above. For other types of request, or if there is no channel associated with the request, it contains blanks.

Note: The routing program's communication area is mapped by the DFHDYPDS DSECT.

Note that the routing program is given the *name* of the channel, not its address, and so is unable to use the DYRCHANL field to inspect or change the contents of the containers.

When a LINK or START command passes a COMMAREA rather than a channel, the routing program can, depending on the type of request, inspect or change the COMMAREA's contents. For LINK requests and transactions started by terminal-related START requests (which are handled by the *dynamic* routing program) but not for non-terminal-related START requests (which are handled by the *distributed* routing program) the routing program is given, in the DYRACMAA field of DFHDYPDS, the *address* of the application's COMMAREA, and can inspect and change its contents.

To give the routing program the same kind of functionality with channels, an application that uses a channel can create, within the channel, a special container named DFHROUTE. If the application issues a LINK or terminal-related START request (but not a non-terminal-related START request) that is to be dynamically routed, the dynamic routing program is given, in the DYRACMAA field of DFHDYPDS, the address of the DFHROUTE container, and can inspect and change its contents.

If you are migrating a program to pass a channel rather than a COMMAREA, you could use its existing COMMAREA structure to map DFHROUTE.

Data conversion

Why is data conversion needed?

Here are some cases in which data conversion is necessary:

- When character data is passed between platforms that use different encoding standards—for example, EBCDIC and ASCII.
- When you want to change the encoding of some character data from one Coded Character Set Identifier (CCSID) to another.

Preparing for code page conversion with channels

About this task

The conversion of data to or from either UTF-8 or UTF-16 and EBCDIC and ASCII codepages, depends on the selection of suitable conversion images. Conversion between the UTF-8 and UTF-16 forms of Unicode is also supported.

Appendix F of the *z/OS Support for Unicode: Using Conversion Services* manual -SA22 -7649 records those conversions which are supported through these services. These are not limited to Unicode, but include the ability to convert between a broad range of character encodings, including EBCDIC, ASCII and Unicode.

Note:

1. The conversion between 037 and 500, as used, for example, with the MQ transport is an EBCDIC to EBCDIC conversion brought about by small differences in the character encodings used by CICS and MQ.

2. You need to be aware that not all points in each codepage have direct counterparts in other codepages. The EBCDIC character NL is one such example. Java and z/OS conversion services may differ in the conversions that they perform. "Technotes", and other Internet discussions may offer guidance on particular points. It is also worth observing that programming communities are themselves divided on the question of what is the more appropriate conversion in particular circumstances.

CICS now supports any of these character conversions by making use of the z/OS conversion services. However, those conversions that earlier releases of CICS carried out using a set of tables, continue to be supported in that manner. It is only if CICS TS 3.1 is asked to carry out a conversion between a pair of CCSIDs that are unsupported via these tables, that it attempts the conversion using the z/OS services.

Ensuring that required conversion images are available

Those CCSIDs used as part of CICS applications must be made known to the System Programmers responsible for maintaining the z/OS Conversion Image, so that specific conversions are available to the CICS regions where these applications execute.

Handling CCSID 1200

CICS supports conversions involving UTF-16 data using any of the following CCSID's: 1200, 1201, and 1202. The z/OS conversion services permit CCSID 1200, in its big-endian form, to be used, but does not contain support for the little-endian form or for CCSIDs 1201 or 1202. CICS transforms any source data that is identified in any of these unsupported forms to the big-endian form of 1200 before passing the data to z/OS for conversion. If the target data is one of the unsupported forms then CICS receives the data as the big-endian form of 1200 and transforms it to the required CCSID. If the target CCSID is 1200 then CICS assumes the encoding to be in big-endian form. If the conversion is between any of these CCSIDs then CICS will carry out the transformation without calling the z/OS conversion services.

When setting up the z/OS conversion image for conversions involving any of these forms of UTF-16 then CCSID 1200 must be specified. CCSIDs 1201 and 1202 will not be recognised by z/OS when attempting to create a conversion image.

CICS respects the byte order marker for inbound conversions, but is not able to retain that information when handling a related outbound conversion. All outbound data for CCSID 1200 is UTF16-BE. Application programmers need to know about this and perform their own BE to LE conversions if they so require.

Sharing a conversion image

- Unless the PTF for APAR OA05744 is applied, do not specify a search order for those conversions, installed into the z/OS image which are intended for use by CICS.
 - If the same conversions are needed for COBOL you must define the conversion image with two separate statements:
 - one with no search order, and
 - the other explicitly specifying a search order of 'RECLM'.
- for example:.

```
CONVERSION 850,037;  
CONVERSION 850,037,RECLM;
```

With the APAR installed, CICS and COBOL can make use of those supported conversions which specify the default search order implicitly or explicitly, removing the need to provide two control statements in the image generation file.

JAVA programs

Codepage conversion facilities exist within JAVA, So it is not necessary to duplicate these in CICS. The conversion facilities described here do not extend to JAVA programs.

Data conversion with channels

Applications that use channels to exchange data use a simple data conversion model. Frequently, no conversion is required and, when it is, a single programming instruction can be used to tell CICS to handle it automatically.

Note the following:

- Usually, when a (non-Java) CICS TS program calls another (non-Java) CICS TS program, no data conversion is required, because both programs use EBCDIC encoding. For example, if a CICS TS C-language program calls a CICS TS COBOL program, passing it some containers holding character data, the only reason for using data conversion would be the unusual one of wanting to change the CCSID of the data.
- The data conversion model used by channel applications is much simpler than that used by COMMAREA applications. Applications that use COMMAREAs to exchange data use the traditional data conversion model described in the *CICS Family: Communicating from CICS on System/390®* manual. Conversion is done under the control of the system programmer, using the DFHCCNV conversion table, the DFHCCNV conversion program and, optionally, the DFHUCNV user-replaceable conversion program.

In contrast, the data in channel containers is converted under the control of the *application* programmer, using API commands.

- The application programmer is responsible only for the conversion of user data—that is, the data in containers created by his application programs. System data is converted automatically by CICS, where necessary.
- The application programmer is concerned only with the conversion of character data. The conversion of binary data (between big-endian and little-endian) is not supported.
- Your applications can use the container API as a simple means of converting character data from one code page to another—see “Using containers to do code page conversion” on page 214.

For data conversion purposes, CICS recognizes two types of data:

CHAR Character data—that is, a text string. The data in the container is converted (if necessary) to the code page of the application that retrieves it. If the application that retrieves the data is a client on an ASCII-based system, this will be an ASCII code page. If it is a CICS Transaction Server for z/OS application, it will be an EBCDIC code page.

All the data in a container is converted as if it were a single character string. For single-byte character set (SBCS) code pages, a structure consisting of several character fields is equivalent to a single-byte character string. However, for double-byte character set (DBCS) code pages this is

not the case. If you use DBCS code pages, to ensure that data conversion works correctly you must put each character string into a separate container.

BIT All non-character data—that is, everything that is not designated as being of type CHAR. The data in the container cannot be converted. This is the default value.

The API commands used for data conversion are:

- ```
EXEC CICS PUT CONTAINER [CHANNEL] [DATATYPE] [FROMCCSID]
```
- ```
EXEC CICS GET CONTAINER [CHANNEL] [INTOCCSID]
```

How to cause CICS to convert character data automatically

About this task

Procedure

1. In the **client program**, use the DATATYPE(DFHVALUE(CHAR)) option of the PUT CONTAINER command to specify that a container holds character data and that the data is eligible for conversion. For example:

```
EXEC CICS PUT CONTAINER(cont_name) CHANNEL('payroll')  
FROM(data1) DATATYPE(DFHVALUE(CHAR))
```

There is no need to specify the FROMCCSID option unless the data is not in the default CCSID of the client platform. (For CICS TS regions, the default CCSID is specified on the LOCALCCSID system initialization parameter.) The default CCSID is implied.

2. In the **server program**, issue a GET CONTAINER command to retrieve the data from the program's current channel:

```
EXEC CICS GET CONTAINER(cont_name) INTO(data_area1)
```

The data is returned in the default CCSID of the server platform. There is no need to specify the INTOCCSID option unless you want the data to be converted to a CCSID other than the default. If the client and server platforms are different, data conversion takes place automatically.

3. In the **server program**, issue a PUT CONTAINER command to return a value to the client:

```
EXEC CICS PUT CONTAINER(status) FROM(data_area2)  
DATATYPE(DFHVALUE(CHAR))
```

The DATATYPE(DFHVALUE(CHAR)) option specifies that the container holds character data and that the data is eligible for conversion. There is no need to specify the FROMCCSID option unless the data is not in the default CCSID of the server platform.

4. In the **client program**, issue a GET CONTAINER command to retrieve the status returned by the server program:

```
EXEC CICS GET CONTAINER(status) CHANNEL('payroll')  
INTO(status_area)
```

The status is returned in the default CCSID of the client platform. There is no need to specify the INTOCCSID option unless you want the data to be converted to a CCSID other than the default. If the client and server platforms are different, data conversion takes place automatically.

Results

Using containers to do code page conversion

About this task

Your applications can use the container API as a simple means of converting character data from one code page to another. The following example converts data from codepage1 to codepage2:

```
EXEC CICS PUT CONTAINER(temp) DATATYPE(DFHVALUE(CHAR))  
           FROMCCSID(codepage1) FROM(input-data)  
EXEC CICS GET CONTAINER(temp) INTOCCSID(codepage2)  
           SET(data-ptr) FLENGTH(data-len)
```

The following example converts data from the region's default EBCDIC code page to a specified UTF8 code page:

```
EXEC CICS PUT CONTAINER(temp) DATATYPE(DFHVALUE(CHAR))  
           FROM(ebcdic-data)  
EXEC CICS GET CONTAINER(temp) INTOCCSID(utf8_ccsid)  
           SET(utf8-data-ptr) FLENGTH(utf8-data-len)
```

When using the container API in this way, bear the following in mind:

- On GET CONTAINER commands, use the SET option, rather than INTO, unless the converted length is known. (You can retrieve the length of the converted data by issuing a GET CONTAINER(*cont_name*) NODATA FLENGTH(*len*) command.)
- To avoid a storage overhead, after conversion copy the converted data and delete the container.
- To avoid shipping the channel, use a temporary channel.
- All-to-all conversion is not possible. That is, a code page conversion error occurs if a specified code page and the channel's code page are an unsupported combination.

A SOAP example

A CICS TS SOAP application:

1. Retrieves a UTF8 or UTF16 message from a socket or MQ message queue.
2. Puts the message into a container, in UTF8 format.
3. Puts EBCDIC data structures into other containers on the same channel.
4. Makes a distributed program link (DPL) call to a handler program on a back-end AOR, passing the channel.

The back-end handler program, also running on CICS TS, can use EXEC CICS GET CONTAINER commands to retrieve the EBCDIC data structures or the messages. It can get the messages in UTF8 or UTF16, or in its own or the region's EBCDIC code page. Similarly, it can use EXEC CICS PUT CONTAINER commands to place data into the containers, in UTF8, UTF16, or EBCDIC.

To retrieve one of the messages in the region's EBCDIC code page, the handler can issue the command:

```
EXEC CICS GET CONTAINER(input_msg) INTO(msg)
```

Because the INTOCCSID option is not specified, the message data is automatically converted to the region's EBCDIC code page. (This assumes that the PUT CONTAINER command used to store the message data in the channel specified a DATATYPE of CHAR; if it specified a DATATYPE of BIT, the default, no conversion is possible.)

To return some output in the region's EBCDIC code page, the handler can issue the command:

```
EXEC CICS PUT CONTAINER(output) FROM(output_msg)
```

Because CHAR is not specified, no data conversion will be permitted. Because the FROMCCSID option is not specified, the message data is taken to be in the region's EBCDIC code page.

To retrieve one of the messages in UTF8, the handler can issue the command:

```
EXEC CICS GET CONTAINER(input_msg) INTO(msg) INTOCCSID(utf8)
```

The INTOCCSID option is necessary to prevent automatic conversion of the data to the region's EBCDIC code page.

To return some output in UTF8, the server program can issue the command:

```
EXEC CICS PUT CONTAINER(output) FROM(output_msg) FROMCCSID(utf8)
```

The FROMCCSID option specifies that the message data is currently in UTF8 format. Because CHAR is not specified, no data conversion will be permitted.

Benefits of channels

The channel/container model has several advantages over the communication areas (COMMAREAs) traditionally used by CICS programs to exchange data. For example:

- Unlike COMMAREAs, channels are not limited in size to 32KB. There is no limit to the number of containers that can be added to a channel, and the size of individual containers is limited only by the amount of storage that you have available.
Take care not to create so many large containers that you limit the amount of
storage available to other applications.
- Because a channel can comprise multiple containers, it can be used to pass data in a more structured way. In contrast, a COMMAREA is a monolithic block of data.
- Unlike COMMAREAs, channels don't require the programs that use them to know the exact size of the data returned.
- A channel is a standard mechanism for exchanging data between CICS programs. A channel can be passed on LINK, START, XCTL, and RETURN commands. Distributed program link (DPL) is supported, and the transactions started by START CHANNEL and RETURN TRANSID commands may be remote.
- Channels can be used by CICS application programs written in any of the CICS-supported languages. For example, a Java client program on one CICS region can use a channel to exchange data with a COBOL server program on a back-end AOR.
- A server program can be written to handle multiple channels. It can, for example:
 1. Discover, dynamically, the channel that it was invoked with
 2. Browse the containers in the channel
 3. Vary its processing according to the channel it's been passed
- You can build “components” from sets of related programs invoked through one or more channels.
- The loose coupling between clients and components permits easy evolution. Clients and components can be upgraded at different times. For example, first a

component could be upgraded to handle a new channel, then the client program upgraded (or a new client written) to use the new channel.

- The programmer is relieved of storage management concerns. CICS automatically destroys containers (and their storage) when they go out of scope.
- The data conversion model used by channel applications is much simpler than that used by COMMAREA applications. Also, whereas in COMMAREA applications data conversion is controlled by the *system* programmer, in channel applications it is controlled by the *application* programmer, using simple API commands.
- Programmers with experience of CICS business transaction services (BTS) will find it easy to use containers in non-BTS applications.
- Programs that use containers can be called from both channel and BTS applications.
- Non-BTS applications that use containers can be migrated into full BTS applications. (They form a migration route to BTS.)

This topic has listed some of the many benefits of channels. However, channels may not be the best solution in all cases. When designing an application, there are one or two implications of using channels that you should be aware of:

- When a channel is to be passed to a remote program or transaction, passing a large amount of data may affect performance. This is particularly true if the local and remote regions are connected by an ISC, rather than MRO, connection.
- A channel may use more storage than a COMMAREA designed to pass the same data. This is because:
 1. Container data can be held in more than one place.
 2. COMMAREAs are accessed by pointer, whereas the data in containers is copied between programs.

Migrating from COMMAREAs to channels

About this task

Migration of existing functions

About this task

- CICS application programs that use traditional communications areas (COMMAREAs) to exchange data continue to work as before.
- If you employ a user-written dynamic or distributed routing program for workload management, rather than CICSplex SM, you must modify your program to handle the new values that it may be passed in the DYRTYPE field of the DFHDYPDS communications area—see the *Customization Guide*.

Migration to the new function

About this task

This section describes how you can migrate several types of existing application to use channels and containers rather than communication areas (COMMAREAs).

It's possible to replace a COMMAREA by a channel with a single container. While this may seem the simplest way to move from COMMAREAs to channels and containers, it's not good practice to do this.

#

Also, be aware that a channel may use more storage than a COMMAREA designed to pass the same data. (See “Benefits of channels” on page 215.)

Because you’re taking the time to change your application programs to exploit this new function, you should implement the “best practices” for channels and containers—see “Designing a channel: best practices” on page 205. Channels have several advantages over COMMAREAs (see “Benefits of channels” on page 215) and it pays to design your channels to make the most of these improvements.

Migrating LINK commands that pass COMMAREAs

About this task

To migrate two programs which use a COMMAREA on a LINK command to exchange a structure, change the instructions shown in Table 13.

Table 13. Migrating LINK commands that pass COMMAREAs

Program	Before	After
PROG1	EXEC CICS LINK PROGRAM(PROG2) COMMAREA(structure)	EXEC CICS PUT CONTAINER(structure-name) CHANNEL(channel-name) FROM(structure) EXEC CICS LINK PROGRAM(PROG2) CHANNEL(channel-name) : EXEC CICS GET CONTAINER(structure-name) CHANNEL(channel-name) INTO(structure)
PROG2	EXEC CICS ADDRESS COMMAREA(structure-ptr) ... RETURN	EXEC CICS GET CONTAINER(structure-name) INTO(structure) ... EXEC CICS PUT CONTAINER(structure-name) FROM(structure) RETURN

Note: In the COMMAREA example, PROG2, having put data in the COMMAREA, has only to issue a RETURN command to return the data to PROG1. In the channel example, to return data *PROG2 must issue a PUT CONTAINER command before the RETURN.*

Migrating XCTL commands that pass COMMAREAs

About this task

To migrate two programs which use a COMMAREA on an XCTL command to pass a structure, change the instructions shown in Table 14.

Table 14. Migrating XCTL commands that pass COMMAREAs

Program	Before	After
PROG1	EXEC CICS XCTL PROGRAM(PROG2) COMMAREA(structure)	EXEC CICS PUT CONTAINER(structure-name) CHANNEL(channel-name) FROM(structure) EXEC CICS XCTL PROGRAM(PROG2) CHANNEL(channel-name) : EXEC CICS GET CONTAINER(structure-name) CHANNEL(channel-name) INTO(structure)
PROG2	EXEC CICS ADDRESS COMMAREA(structure-ptr) ...	EXEC CICS GET CONTAINER(structure-name) INTO(structure) ...

Migrating pseudoconversational COMMAREAs on RETURN commands

About this task

To migrate two programs which use COMMAREAs to exchange a structure as part of a pseudoconversation, change the instructions shown in Table 15.

Table 15. Migrating pseudoconversational COMMAREAs on RETURN commands

Program	Before	After
PROG1	EXEC CICS RETURN TRANSID(PROG2) COMMAREA(structure)	EXEC CICS PUT CONTAINER(structure-name) CHANNEL(channel-name) FROM(structure) EXEC CICS RETURN TRANSID(TRAN2) CHANNEL(channel-name)
PROG2	EXEC CICS ADDRESS COMMAREA(structure-ptr)	EXEC CICS GET CONTAINER(structure-name) INTO(structure)

Migrating START data

About this task

To migrate two programs which use START data to exchange a structure, change the instructions shown in Table 16.

Table 16. Migrating START data

Program	Before	After
PROG1	EXEC CICS START TRANSID(TRAN2) FROM(structure)	EXEC CICS PUT CONTAINER(structure-name) CHANNEL(channel-name) FROM(structure) EXEC CICS START TRANSID(TRAN2) CHANNEL(channel-name)
PROG2	EXEC CICS RETRIEVE INTO(structure)	EXEC CICS GET CONTAINER(structure-name) INTO(structure)

Note that the new version of PROG2 is the same as that in the pseudoconversational example.

Migrating programs that use temporary storage to pass data

About this task

In previous releases, because the size of COMMAREAs is limited to 32K and channels were not available, some applications used temporary storage queues (TSQs) to pass more than 32K of data from one program to another. Typically, this involved multiple writes to and reads from a TSQ.

If you migrate one of these applications to use channels, be aware that:

- If the TS queue used by your existing application is in main storage, the storage requirements of the new, migrated, application are likely to be similar to those of the existing application.
- If the TS queue used by your existing application is in auxiliary storage, the storage requirements of the migrated application are likely to be greater than those of the existing application. This is because container data is held in storage rather than being written to disk.

Migrating dynamically-routed applications

EXEC CICS LINK and EXEC CICS START commands, which can pass either COMMAREAs or channels, can be dynamically routed.

When a LINK or START command passes a COMMAREA rather than a channel, the routing program can, depending on the type of request, inspect or change the COMMAREA's contents. For LINK requests and transactions started by terminal-related START requests (which are handled by the *dynamic* routing program) but not for non-terminal-related START requests (which are handled by the *distributed* routing program) the routing program is given, in the DYRACMAA field of its communication area, the *address* of the application's COMMAREA, and can inspect and change its contents.

Note: The routing program's communication area is mapped by the DFHDYPDS DSECT.

If you migrate a dynamically-routed EXEC CICS LINK or START command to use a channel rather than a COMMAREA, the routing program is passed, in the DYRCHANL field of DFHDYPDS, the name of the channel. Note that the routing program is given the *name* of the channel, not its address, and so is unable to use the DYRCHANL field to inspect or change the contents of the channel's containers.

To give the routing program the same kind of functionality with channels, an application that uses a channel can create, within the channel, a special container named DFHROUTE. If the application issues a LINK or terminal-related START request (but not a non-terminal-related START request) that is to be dynamically routed, the dynamic routing program is given, in the DYRACMAA field of DFHDYPDS, the address of the DFHROUTE container, and can inspect and change its contents.

If you are migrating a program to pass a channel rather than a COMMAREA, you could use its existing COMMAREA structure to map DFHROUTE.

Chapter 15. Affinity

CICS transactions and programs use many different techniques to pass data from one to another. Some of these techniques require that the transactions or programs exchanging data must execute in the same CICS region. This imposes restrictions on the regions to which transactions and distributed program link (DPL) requests can be dynamically routed. If transactions or programs exchange data in ways that impose such restrictions, there is said to be an affinity among them.

Java

This guidance on affinity between transactions describes applications written using the EXEC CICS API. However, many of the comments are equally valid for Java applications and enterprise beans executing in a CICSplex. For guidance on developing Java applications and enterprise beans, see *Java Applications in CICS*.

Transactions, program-link requests, EXEC CICS START requests, CICS business transaction services (BTS) activities, and enterprise bean method calls can all be dynamically routed.

You can use a *dynamic* routing program to dynamically route:

- Transactions started from terminals
- Transactions started by eligible terminal-related EXEC CICS START commands
- Eligible CICS-to-CICS DPL requests
- Eligible program-link requests received from outside CICS.

You can use a *distributed* routing program to dynamically route:

- Eligible BTS processes and activities. (BTS is described in the *CICS Business Transaction Services* manual.)
- Eligible non-terminal-related EXEC CICS START requests.

For detailed introductory information about dynamic and distributed routing, see the *CICS Intercommunication Guide*.

Important

The following sections talk exclusively about affinities between *transactions*. Keep in mind throughout the chapter that:

- Affinities can also exist between programs. (Although, strictly speaking, we could say that it is the transactions associated with the programs that have the affinity.) This may impose restrictions on the regions to which program-link requests can be routed.
- The sections on safe, unsafe, and suspect programming techniques apply to the routing of program-link and START requests, as well as to the routing of transactions.

This chapter describes:

- “Types of affinity” on page 222
- “Programming techniques and affinity” on page 223
- “Safe programming to avoid affinity” on page 224
- “Unsafe programming for affinity” on page 229

- “Suspect programming for affinity” on page 236
- “Detecting inter-transaction affinities” on page 246
- “Duration and scope of inter-transaction affinities” on page 246

Types of affinity

There are two types of affinity that affect dynamic routing:

- Inter-transaction affinity
- Transaction-system affinity

Inter-transaction affinity

Transaction affinity among two or more CICS transactions is caused by the transactions using techniques to pass information between one another, or to synchronize activity between one another, in a way that requires the transactions to execute in the same CICS region. This type of affinity is inter-transaction affinity, where a set of transactions share a common resource and/or coordinate their processing. Inter-transaction affinity, which imposes restrictions on the dynamic routing of transactions, can occur in the following circumstances:

- One transaction terminates, leaving ‘state data’ in a place that a second transaction can only access by running in the same CICS region as the first transaction.
- One transaction creates data that a second transaction accesses while the first transaction is still running. For this to work safely, the first transaction usually waits on some event, which the second transaction posts when it has read the data created by the first transaction. This technique requires that both transactions are routed to the same CICS region.
- Two transactions synchronize, using an event control block (ECB) mechanism. Because CICS has no function shipping support for this technique, this type of affinity means the two transactions must be routed to the same CICS region.

Note: The same is true if two transactions synchronize, using an enqueue (ENQ) mechanism, **unless** you have used appropriate ENQMODEL resource definitions to give sysplex-wide scope to the ENQs. See the *CICS Resource Definition Guide* .

Transaction-system affinity

There is another type of transaction affinity that is not an affinity among transactions themselves. It is an affinity between a transaction and a particular CICS region, where the transaction interrogates or changes the properties of that CICS region—transaction-system affinity.

Transactions with affinity to a particular system, rather than another transaction, are not eligible for dynamic routing. In general, they are transactions that use INQUIRE and SET commands, or have some dependency on global user exit programs, which also have an affinity with a particular CICS region.

Using INQUIRE and SET commands and global user exits

There is no remote (that is, function shipping) support for INQUIRE and SET commands, nor is there a SYSID option on them, hence transactions using these commands must be routed to the CICS regions that own the resources to which they refer. In general, such transactions cannot be dynamically routed to any target region, and therefore transactions that use INQUIRE and SET should be statically routed.

Global user exits running in different CICS regions cannot exchange data. It is unlikely that user transactions pass data or parameters by means of user exits, but if such transactions do exist, they must run in the same target region as the global user exits.

Programming techniques and affinity

From the point of view of inter-transaction affinity in a dynamic or distributed routing environment, the programming techniques used by your application programs can be considered in three broad categories:

- Those techniques that are generally safe and do not cause inter-transaction affinities
- Those techniques that are inherently unsafe
- Those techniques that are suspect in that they may, or may not, create affinities depending on exactly how they are implemented

Safe techniques

The programming techniques in the generally safe category are:

- The use of the communication area (COMMAREA), supported by the CICS API on a number of CICS commands. However, it is the COMMAREA option on the CICS RETURN command only that is of interest in a dynamic or distributed routing environment with regard to transaction affinity, because it is the COMMAREA on a RETURN command that is passed to the next transaction in a pseudoconversational transaction.
- The use of a TCT user area (TCTUA) that is optionally available for each terminal defined to CICS.
- Synchronization or serialization of tasks using CICS commands:
 - ENQ / DEQ, **provided that** you have used appropriate ENQMODEL resource definitions to give sysplex-wide scope to the ENQs. See “Using ENQ and DEQ commands with ENQMODEL resource definitions” on page 228 and the *CICS Resource Definition Guide* for a description of ENQMODELS.
- The use of containers to pass data between CICS Business Transaction Services (BTS) activities. Container data is saved in an RLS-enabled BTS VSAM file.

For more information about the COMMAREA and the TCTUA, see “Safe programming to avoid affinity” on page 224.

Unsafe techniques

The programming techniques in the unsafe category are:

- The use of long-life shared storage:
 - The common work area (CWA)
 - GETMAIN SHARED storage
 - Storage obtained by a LOAD PROGRAM HOLD
- The use of task-lifetime local storage shared by synchronized tasks

It is possible for one task to pass the address of some task-lifetime storage to another task.

It may be safe for the receiving task to use the passed address, provided the owning task does not terminate. It is possible, but ill-advised, to use a CICS task-synchronization technique to allow the receiving task to prevent the sending task from terminating (or freeing the storage in some other way) before the receiver has finished with the address. However, such designs are not robust because there is a danger of the sending task being purged by some outside agency.

See “Sharing task-lifetime storage” on page 232 for more details.

- Synchronization or serialization of tasks using CICS commands:
 - WAIT EVENT / WAIT EXTERNAL / WAITCICS
 - ENQ / DEQ, **unless** you have used appropriate ENQMODEL resource definitions to give sysplex-wide scope to the ENQs. See “Using ENQ and DEQ commands with ENQMODEL resource definitions” on page 228 and the *CICS Resource Definition Guide* for a description of ENQMODELS.

For more information about unsafe programming techniques, see “Unsafe programming for affinity” on page 229.

Suspect techniques

Some programming techniques may, or may not, create affinity depending on exactly how they are implemented. A good example is the use of temporary storage. Application programs using techniques in this category must be checked to determine whether they work without restrictions in a dynamic or distributed routing environment. The programming techniques in the suspect category are:

- The use of temporary storage queues with restrictive naming conventions
- Transient data queues and the use of trigger levels
- Synchronization or serialization of tasks using CICS commands:
 - RETRIEVE WAIT / START
 - START / CANCEL REQID
 - DELAY / CANCEL REQID
 - POST / CANCEL REQID
- INQUIRE and SET commands and global user exits

For more information about suspect programming techniques, see “Suspect programming for affinity” on page 236.

Recommendations

The best way to deal with inter-transaction affinity is to avoid creating inter-transaction affinity in the first place.

Where it is not possible to avoid affinities, you should:

- Make the inter-transaction affinity easily recognizable, by using appropriate naming conventions, and
- Keep the lifetime of the affinities as short as possible.

Even if you could avoid inter-transaction affinities by changing your application programs, this is not absolutely necessary provided you include logic in your dynamic and distributed routing programs to cope with the affinities. Finally, you can statically route the affected transactions.

Safe programming to avoid affinity

Some techniques for passing data between transactions are generally safe in that they do not create inter-transaction affinity. These involve the use of a communication area (COMMAREA), a terminal control table user area (TCTUA), or BTS containers.

However, to remain free from affinity, COMMAREAs, TCTUAs, and BTS containers must **not** contain addresses. Generally the storage referenced by such addresses

would have to be long-life storage, the use of which is an unsafe programming technique in a dynamic transaction routing environment.

The COMMAREA

The use of the COMMAREA option on the RETURN command is the principal example of a safe programming technique that you can use to pass data between successive transactions in a CICS pseudoconversational transaction. CICS treats the COMMAREA as a special form of user storage, even though it is CICS that issues the GETMAIN and FREEMAIN requests for the storage, and not the application program.

CICS ensures that the contents of the COMMAREA specified on a RETURN command are always made available to the first program in the next transaction. This is true even when the sending and receiving transactions execute in different target regions. In a pseudoconversation, regardless of the target region to which a dynamic routing program chooses to route the next transaction, CICS ensures the COMMAREA specified on the previous RETURN command is made available in the target region. This is illustrated in Figure 54 on page 226.

Some general characteristics of a COMMAREA are:

- Processor overhead is low.
- It is not recoverable.
- The length of a COMMAREA on a RETURN command can vary from transaction to transaction, up to a theoretical upper limit of 32 763 bytes. (However to be safe, you should not exceed 24KB (1KB = 1024 bytes), as recommended in the Application Programming Reference manual, because of a number of factors that can reduce the limit from the theoretical maximum.)
- CICS holds a COMMAREA in CICS main storage until the terminal user responds with the next transaction. This may be an important consideration if you are using large COMMAREAs, because the number of COMMAREAs held by CICS relates to terminal usage, and not to the maximum number of tasks in a region at any one time.
- A COMMAREA is available only to the first program in the next transaction, unless that program explicitly passes the data to another program or a succeeding transaction.

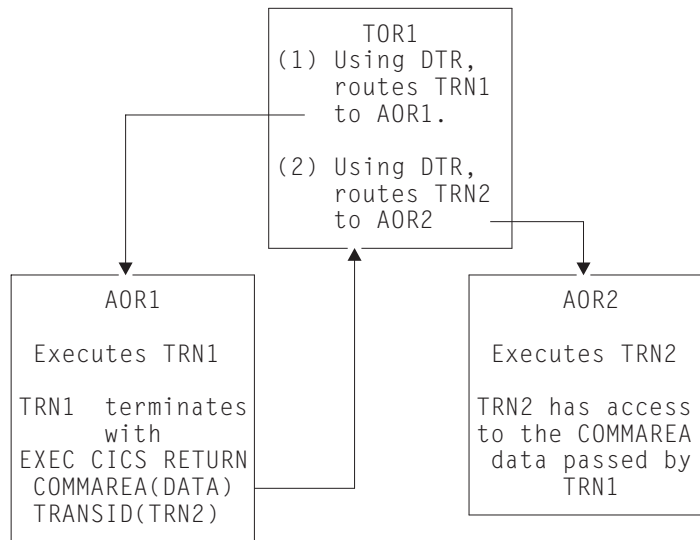


Figure 54. The use of a COMMAREA by a pseudoconversation in a dynamic transaction routing environment

The COMMAREA used in a pseudoconversational transaction, as shown in Figure 54, can be passed from transaction to transaction across a CICSplex, and, provided the COMMAREA contains only data and not addresses of storage areas, no inter-transaction affinity is created.

The TCTUA

The TCTUA is an optional extension to the terminal control table entry (TCTTE), each entry specifying whether the extension is present, and its length. You specify that you want a TCTUA associated with a terminal by defining its length on the USERAREALEN parameter of a TYPETERM resource definition. This means that the TCTUAs are of fixed length for all the terminals created using the same TYPETERM definition.

A terminal control table user area (TCTUA) is safe to use in a dynamic transaction routing environment as a means of passing data between successive transactions in a pseudoconversational transaction. Like the COMMAREA, the TCTUA is always accessible to transactions initiated at a user terminal, even when the transactions in a pseudoconversation are routed to different target regions. This is illustrated in Figure 55 on page 227. Some other general characteristics of TCTUAs are:

- Minimal processor overhead (only one CICS command is needed to obtain the address).
- It is not recoverable.
- The length is fixed for the group of terminals associated with a given TYPETERM definition. It is suitable only for small amounts of data, the maximum size allowed being 255 bytes.
- If the terminal is autoinstalled, the TCTUA lasts as long as the TCTTE, the retention of which is determined by the AILDELAY system initialization parameter. The TCTTE, and therefore any associated TCTUA, is deleted when the AILDELAY interval expires after a session between CICS and a terminal is ended.

If the terminal is defined to CICS by an explicit terminal definition, the TCTTE and its associated TCTUA are created when the terminal is installed and remain until the next initial or cold start of CICS.

Note that the TCTUA is available to a dynamic routing environment in the routing region as well as application programs in the target region. It can be used store information relating to the dynamic routing of a transaction. For example, you can use the TCTUA to store the name of the selected target region to which a transaction is routed.

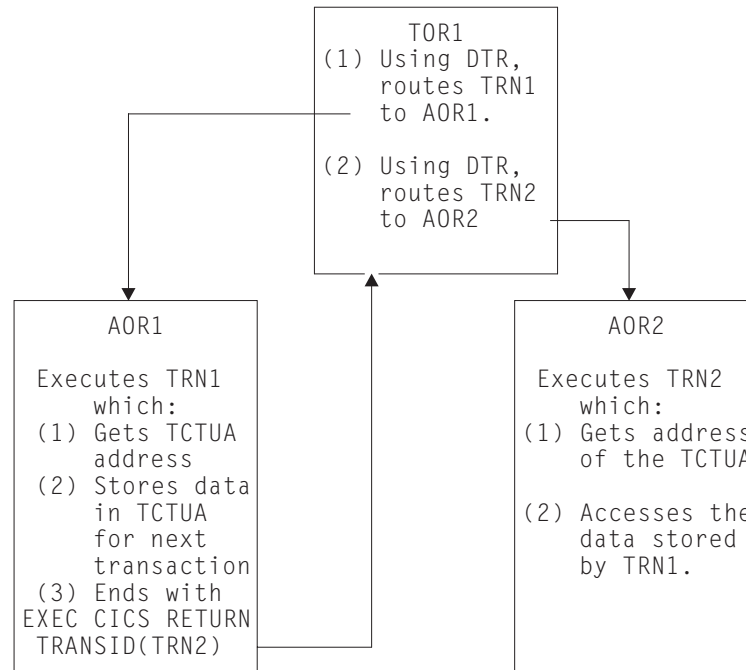


Figure 55. The use of a TCTUA by a pseudoconversation in a dynamic routing environment

Using the TCTUA in an unsafe way

The EXEC CICS ADDRESS TCTUA(*ptr-ref*) provides direct addressability to the TCTUA, and this is how each task requiring access to a TCTUA should obtain the TCTUA address. If tasks attempt to pass the address of their TCTUAs in some other way, such as in a temporary storage queue, or to use the TCTUA itself to pass addresses of other storage areas, the TCTUA ceases to provide a safe programming technique for use in a dynamic transaction routing environment.

It is also possible for a task to obtain the TCTUA of a principal facility other than its own, by issuing an INQUIRE TERMINAL command that names the terminal associated with another task (the INQUIRE TERMINAL command returns the TCTUA address of the specified terminal). Using the TCTUA address of a terminal other than a task's own principal facility is another example an unsafe use of the TCTUA facility. Depending on the circumstances, particularly in a dynamic routing environment, the TCTUA of a terminal that is not the inquiring task's principal facility could be deleted after the address has been obtained. For example, in an target region, an INQUIRE TERMINAL command could return the TCTUA address associated with a surrogate terminal that is running a dynamically routed transaction. If the next transaction from the terminal is routed to a different target region, the TCTUA address ceases to be valid.

Using ENQ and DEQ commands with ENQMODEL resource definitions

The ENQ and DEQ commands are used to serialize access to a shared resource. In earlier releases of CICS, these commands were limited to the scope of CICS tasks running in the same region, and could not be used to serialize access to a resource shared by tasks in different regions. Now, **provided that** the ENQs and DEQs are supported by appropriate ENQMODEL resource definitions (see the *CICS Resource Definition Guide* for a description of ENQMODELS) they can have sysplex-wide scope.

This is primarily of interest to the system programmer who will determine transaction routing decisions, but application programmers should be aware of the advantages now available.

Overview of sysplex enqueue and dequeue

Changes to the CICS enqueue/dequeue function extend the CICS application programming interface to provide an enqueue mechanism that serializes access to a **named resource** across a specified set of CICS regions operating within a sysplex. This applies equally to a CICSplex within a single MVS image and to a CICSplex that resides in more than one MVS. (Note that sysplex-wide enqueue is supported only for a **resource**, and not for an enqueue on an **address**.)

Local enqueues within a single CICS region are managed within the CICS address space. Sysplex-wide enqueues that affect more than one CICS region are managed by Global Resource Services (GRS). The main points of the changes to the CICS enqueue/dequeue mechanism are as follows:

- Sysplex enqueue and dequeue expands the scope of an EXEC CICS ENQIDEQ command from region to sysplex, by introducing a new CICS resource definition type, ENQMODEL, to define resource names that are to be sysplex-wide.
- ENQSCOPE, an attribute of the ENQMODEL resource definition, defines the set of regions that share the same enqueue scope.
- When an EXEC CICS ENQ (or DEQ) command is issued for a resource whose name matches that of an installed ENQMODEL resource definition, CICS checks the value of the ENQSCOPE attribute to determine whether the scope is local or sysplex-wide, as follows:
 - If the ENQSCOPE attribute is left blank (the default value), CICS treats the ENQIDEQ as local to the issuing CICS region.
 - If the ENQSCOPE is non-blank, CICS treats the ENQIDEQ as sysplex-wide, and passes a queue name and the resource name to GRS to manage the enqueue. The resource name is as specified on the EXEC CICS ENQIDEQ command, and the queue name is made up by prefixing the 4-character ENQSCOPE with the letters DFHE.
- The CICS regions that need to use sysplex-wide enqueue/dequeue function must all have the required ENQMODELS defined and installed.

The recommended way to ensure this is for the CICS regions to share a CSD, and for the initialization group lists to include the same ENQMODEL groups.

Existing applications can use sysplex enqueues simply by defining appropriate ENQMODELS, without any change to the application programs.

Benefits

Sysplex enqueue provides the following benefits:

- Eliminates one of the most common causes of inter-transaction affinity.
- Enables better exploitation of a parallel sysplex providing better price/performance, capacity, and availability.
- Reduces the need for inter-transaction affinity rules in dynamic and distributed routing programs thereby lowering the systems management cost of exploiting parallel sysplex.
- Enables serialization of concurrent updates to shared temporary storage queues, performed by multiple CICS tasks across the sysplex.
- Makes it possible to prevent interleaving of records written by concurrent tasks in different CICS regions to a remote transient data queue.
- Allows the single-threading and synchronization of tasks across the sysplex. It is not designed for the locking of recoverable resources.

BTS containers

A container is owned by a BTS activity. Containers cannot be used outside of an activity; for more information, see the *CICS Business Transaction Services* manual . A container may be used to pass data between BTS activities or between different activations of the same activity. An activity uses GET and PUT container to update the container's contents. CICS ensures that the appropriate containers are available to an activity by saving all the information (including containers) associated with a BTS activity in an RLS-enabled VSAM file. For this reason, note that a BTS environment cannot extend outside a sysplex (see *CICS Business Transaction Services*), but you can use dynamic routing within a sysplex passing data in containers.

Some general characteristics of containers are:

- An activity may own any number of containers; you are not limited to one.
- There is no size restriction.
- They are recoverable.
- They exist in main storage only while the associated activity is executing. Otherwise they are held on disk. Therefore, you do not need to be overly concerned with their storage requirements, unlike terminal COMMAREAs.

Unsafe programming for affinity

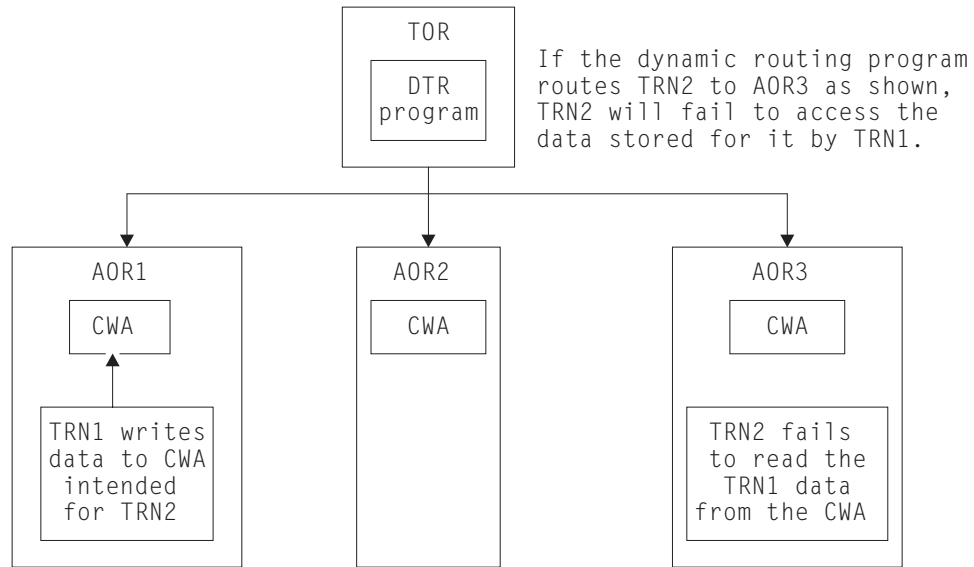
Some CICS application programming techniques, notably those that pass, or obtain, addresses to shared storage, create an affinity between transactions.

The programming techniques that are generally unsafe are described in the following sections.

Using the common work area

The CWA in a CICS region is created (optionally) during CICS initialization, exists until CICS terminates, and is not recovered on a CICS restart (warm or emergency). The ADDRESS CWA(*ptr-ref*) command provides direct addressability to the CWA.

A good example of how the use of long-life shared storage such as the CWA can create affinity is when one task stores data in the CWA, and a later task reads the data from it. Clearly, the task retrieving the data must run in the same target region as the task that stored the data, or it references a completely different storage area in a different address space. This restricts the workload balancing capability of the dynamic or distributed routing program, as shown in Figure 56.



CWA

Figure 56. Illustration of inter-transaction affinity created by use of the CWA. The dynamic routing program needs to be aware of this CWA affinity, and ensure it routes TRN2 to the same target region as TRN1.

However, if the CWA contains read-only data, and this data is replicated in more than one target region, it is possible to use the CWA and continue to have the full benefits of dynamic routing. For example, you can run a program during the post-initialization phase of CICS startup (a PLTPI program) that loads the CWA with read-only data in each of a number of selected target regions. In this way, all transactions routed to target regions loaded with the same CWA data have equal access to the data, regardless of which of the target regions to which the transactions are routed. With CICS subsystem storage protection, you can ensure the read-only integrity of the CWA data by requesting the CWA from CICS-key storage, and define all the programs that read the CWA to execute in user key.

Using GETMAIN SHARED storage

Shared storage is allocated by a GETMAIN SHARED command, and remains allocated until explicitly freed by the same, or by a different, task. Shared storage can be used to exchange data between any CICS tasks that run during the lifetime of the shared storage. Transactions designed in this way must execute in the same CICS region to work correctly. The dynamic or distributed routing program should ensure that transactions using shared storage are routed to the same target region.

Figure 57 on page 231 illustrates the use of shared storage.

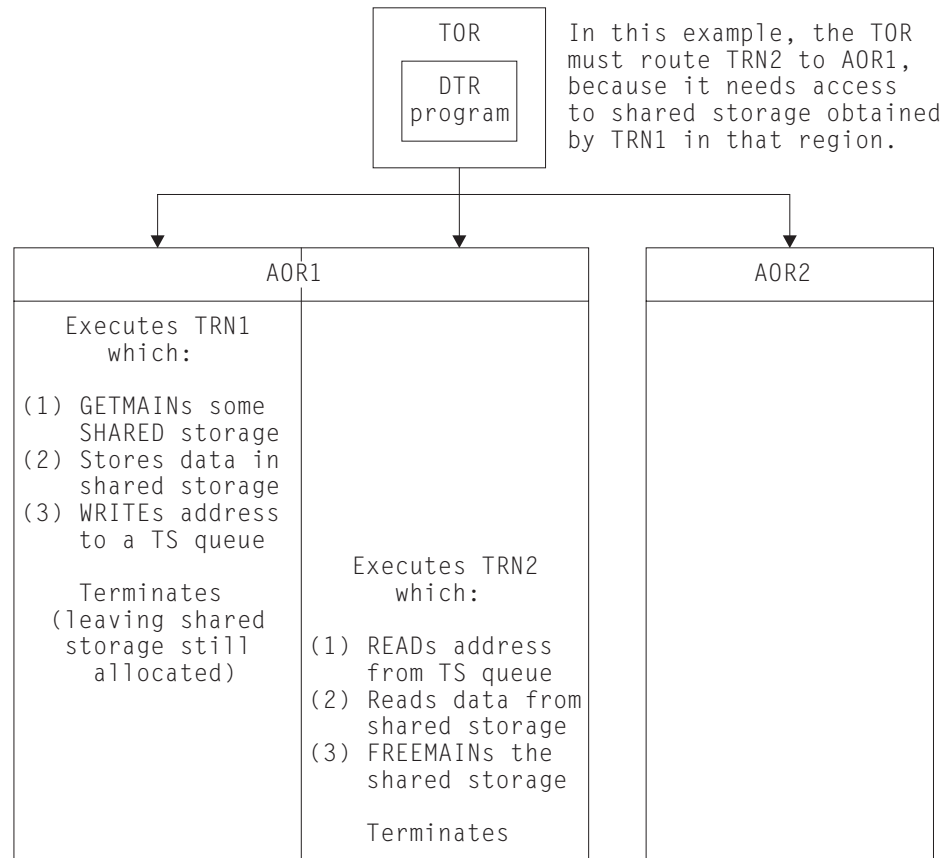


Figure 57. Illustration of inter-transaction affinity created by use of shared storage. The dynamic transaction routing program needs to be aware of this affinity, and ensure it routes TRN2 to the same target region as TRN1.

If the two transactions shown in Figure 57 are parts of a pseudoconversational transaction, the use of shared storage should be replaced by a COMMAREA (provided that the amount of storage fits within the COMMAREA size limits).

Using the LOAD PROGRAM HOLD command

A program (or table) that CICS loads in response to a LOAD PROGRAM HOLD command remains in directly addressable storage until explicitly released by the same, or by a different, task. Any CICS tasks that run while the loaded program (table) is held in storage can use the loaded program's storage to exchange data, provided that:

- The program is not loaded into read-only storage, or
- The program is not defined to CICS with RELOAD(YES)

Although you could use a temporary storage queue to make the address of the loaded program's storage available to other tasks, the more usual method would be for other tasks to issue a LOAD PROGRAM command also, with the SET(*ptr_ref*) option so that CICS can return the address of the held program.

The nature of the affinity caused by the use of the LOAD PROGRAM HOLD command is virtually identical to that caused by the use of GETMAIN SHARED storage (see Figure 57 and Figure 58 on page 232), and the same rule applies: to preserve the application design, the dynamic or distributed routing program must ensure that all transactions that use the address of the loaded program (or table)

are routed to the same target region.

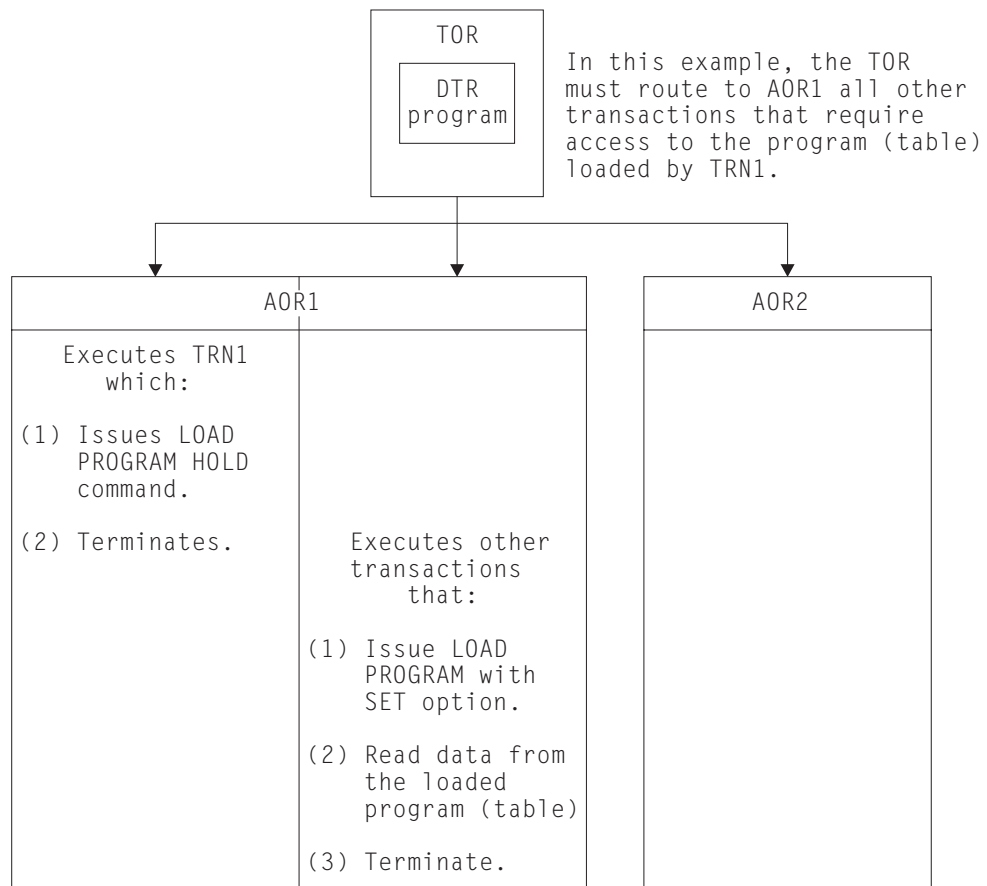


Figure 58. Illustration of inter-transaction affinity created by use of LOAD PROGRAM HOLD. The dynamic routing program needs to be aware of this affinity, and ensure it routes TRN2 to the same target region as TRN1.

Note: This rule applies also to programs defined with the RESIDENT option on the resource definition for the loaded program (whether or not the HOLD option is specified on the LOAD command). However, regardless of affinity considerations, it is unsafe to use the RESIDENT option to enable transactions to share data, because programs defined with RESIDENT are subject to SET PROGRAM(*program_name*) NEWCOPY commands, and can therefore be changed.

The rule also applies to a non-resident, non-held, loaded program where the communicating tasks are synchronized.

Sharing task-lifetime storage

The use of any task-lifetime storage belonging to one task can be shared with another task, provided the owning task can pass the address to the other task in the same CICS address space. This technique creates an affinity among the communicating tasks, and requires that any task retrieving and using the passed address must execute in the same target region as the task owning the task-lifetime storage.

For example, it is possible to use a temporary storage queue to pass the address of a PL/I automatic variable, or the address of a COBOL working-storage structure

(see Figure 59 for an example).

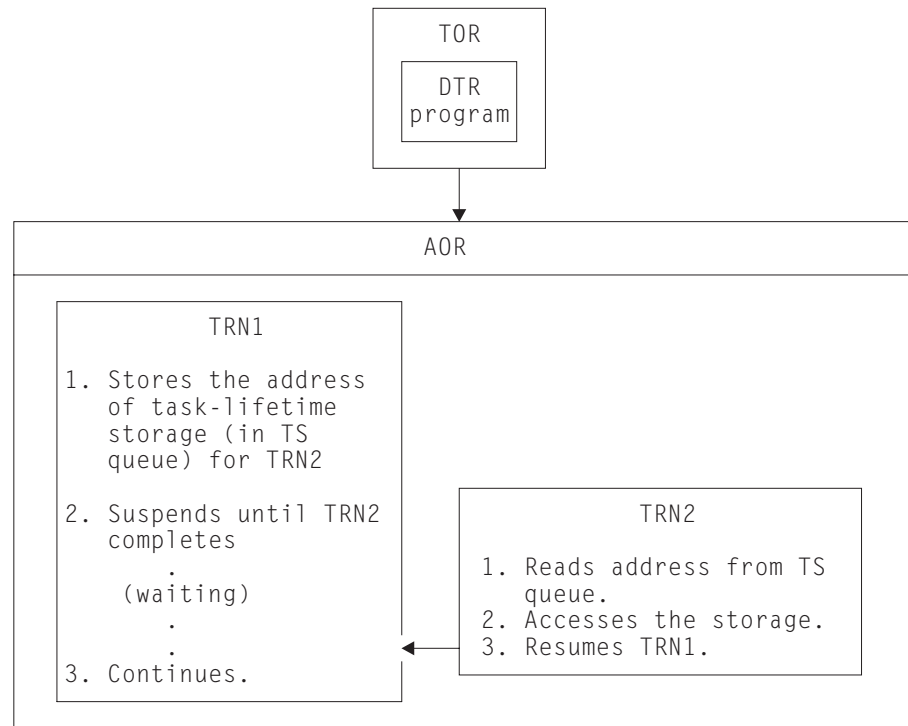


Figure 59. Illustration of inter-transaction affinity created by use of task-lifetime storage. TRN2 must execute in the same target region as TRN1. Also, TRN1 must not terminate until TRN2 has finished using its task-lifetime storage.

For two tasks to share task-lifetime storage belonging to one of them requires that the tasks are synchronized in some way. See Table 17 for commands that provide ways of suspending and resuming a task that passes the address of its local storage.

Table 17. Methods for suspending and resuming (synchronizing) tasks

Suspending operation	Resuming operation
WAIT EVENT, WAIT EXTERNAL, WAITCICS	POST
RETRIEVE WAIT	START
DELAY	CANCEL
POST	CANCEL
START	CANCEL
ENQ	DEQ

Some of these techniques themselves require that the transactions using them must execute in the same target region, and these are discussed later in this chapter. However, even in those cases where tasks running in different target regions can be synchronized, it is not safe to pass the address of task-lifetime storage from one to the other. Even without dynamic routing, designs that are based on the synchronization techniques shown in Table 17 are fundamentally unsafe because it is possible that the storage-owning task could be purged.

Notes:

1. Using synchronization techniques, such as RETRIEVE WAIT/START, to allow sharing of task-lifetime storage is unsafe in CICS Version 2 because the task issuing, for example, the RETRIEVE WAIT could be purged by a CEMT SET TASK(...) PURGE command. In CICS/ESA Version 3 and later, the SPURGE parameter on the transaction definition could be used to protect the first task, but even so the design is not recommended.
2. No inter-transaction affinity is caused in those cases where the task sharing another task's task-lifetime storage is started by an START command, except when the START command is function-shipped or routed to a remote system.

Using the WAIT EVENT command

The WAIT EVENT command is used to synchronize a task with the completion of an event performed by some other CICS or MVS task.

The completion of the event is signalled (posted) by the setting of a bit pattern into the event control block (ECB). Both the waiting task and the posting task must have direct addressability to the ECB, hence both tasks must execute in the same target region. The use of a temporary storage queue is one way that the waiting task can pass the address of the ECB to another task.

This synchronization technique is illustrated in Figure 60.

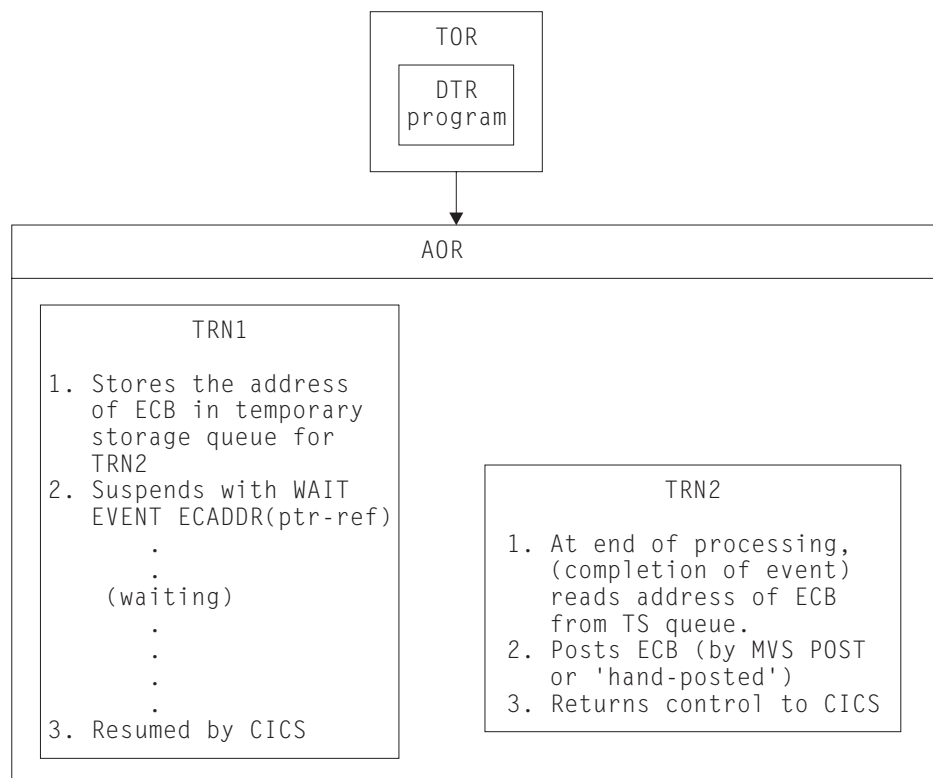


Figure 60. Illustration of inter-transaction affinity created by use of WAIT EXTERNAL command. TRN2 must execute in the same target region as TRN1.

If TRN2 shown in Figure 60 executed in a different target region from TRN1, the value of *ptr-ref* would be invalid, the post operation would have unpredictable results, and the waiting task would never be resumed. For this reason, a dynamic

or distributed routing program must ensure that a posting task executes in the same target region as the waiting task to preserve the application design. The same considerations apply to the use of WAIT EXTERNAL and WAITCICS commands for synchronizing tasks.

Using ENQ and DEQ commands without ENQMODEL resource definitions

The ENQ and DEQ commands are used to serialize access to a shared resource. These commands only work for CICS tasks running in the same region, and cannot be used to serialize access to a resource shared by tasks in different regions, **unless** they are supported by appropriate ENQMODEL resource definitions so that they have sysplex-wide scope. See “Using ENQ and DEQ commands with ENQMODEL resource definitions” on page 228 and the *CICS Resource Definition Guide* for a description of ENQMODELS.

Note that any ENQ that does not specify the LENGTH option is treated as an enqueue on an **address** and therefore has only local scope. The use of ENQ and DEQ for serialization (without ENQMODEL definitions to give sysplex-wide scope) is illustrated in Figure 61.

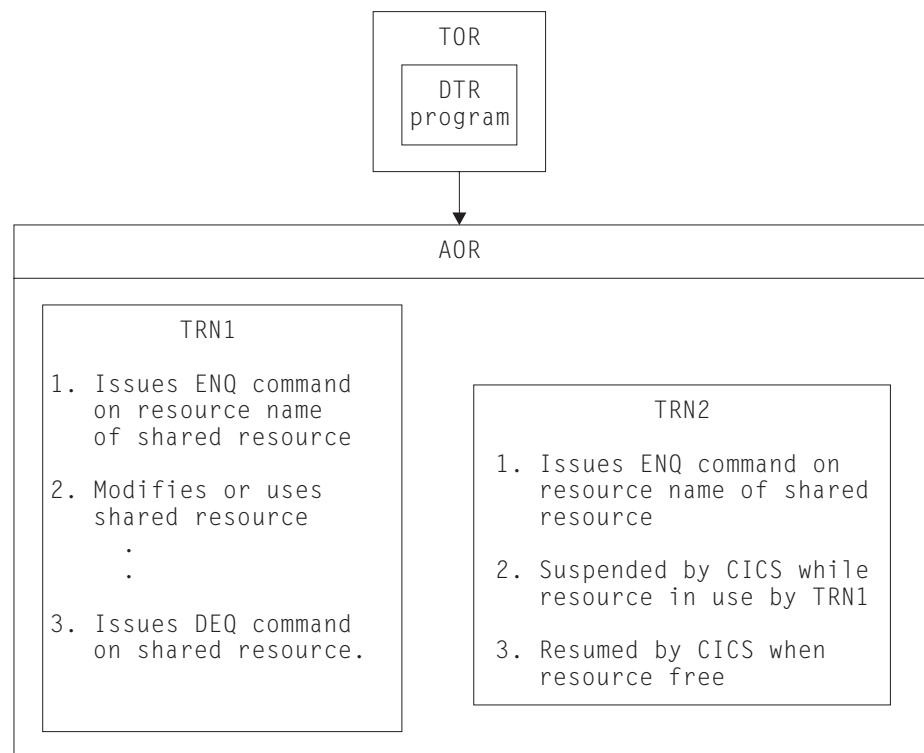


Figure 61. Illustration of inter-transaction affinity created by use of ENQ/DEQ commands. TRN2 must execute in the same target region as TRN1.

If TRN2 shown in Figure 61 executed in a different target region from TRN1, TRN2 would not be suspended while TRN1 accessed the shared resource. For this reason, a dynamic or distributed routing program must ensure that all tasks that enqueue on a given resource name must execute in the same target region to preserve the application design. TRN2 would, of course, be serialized with other CICS tasks that issue ENQ commands on the same resource name in its target region.

Suspect programming for affinity

Some CICS application programming techniques may create an affinity between transactions depending on how they are implemented.

The programming techniques that may be suspect are described in the following sections.

Using temporary storage

CICS application programs commonly use temporary storage (TS) queues to hold temporary application data, and to act as scratch pads.

Sometimes a TS queue is used to pass data between application programs that execute under one instance of a transaction (for example, between programs that pass control by a LINK or XCTL command in a multi-program transaction). Such use of a TS queue requires that the queue exists only for the lifetime of the transaction instance, and therefore it does not need to be shared between different target regions, because a transaction instance executes in one, and only one, target region.

Note: This latter statement is not strictly true in the case of a multi-program transaction, where one of the programs is linked by a distributed program link command and the linked-to program resides in a remote system. In this case, the program linked by a DPL command runs under another CICS task in the remote region. The recommended method for passing data to a DPL program is by a COMMAREA, but if a TS queue is used for passing data in a DPL application, the queue must be shared between the two regions.

Sometimes a TS queue holds information that is specific to the target region, or holds read-only data. In this case the TS queue can be replicated in each target region, and no sharing of data between target regions is necessary.

However, in many cases a TS queue is used to pass data between transactions, in which case the queue must be globally accessible to enable the transactions using the queue to run in any dynamically selected target region. It is possible to make a temporary storage queue globally accessible by function shipping TS requests to a queue-owning region (QOR), provided the TS queue can be defined as remote. Shared queues are defined by using a temporary storage pool in a coupling facility. Shared temporary storage applies only to non-recoverable queues. You can make queues in auxiliary storage recoverable, but not queues in main storage.

In a pseudoconversational transaction, you can change the program to use a COMMAREA to pass data between the phases of the conversation. However, using temporary storage data-sharing avoids inter-transaction affinity by being able to use dynamic routing to any target region. Shared temporary storage queue requests for specific SYSIDs are routed in the same way as remote queue requests. The SYSID value defined to shared TS pools is TST TYPE=SHARED.

The methods for specifying TS pool make it easy to migrate queues from a QOR to a TS data-sharing pool. You can use the temporary storage global user exit, XTSEREQ, to modify the SYSID on a TS request so that it references a TS data-sharing pool. Figure 62 shows four AORs that are using the same TST (TST=XX). The SYSIDNT option on the TST TYPE=SHARED macro maps temporary storage requests to a pool of shared TS queues called DFHXQTS1. You should define DFHXQTS1 in the poolname parameter of the JCL to start up the TS

data sharing server DFHXQMN. See the *CICS System Definition Guide* for more information about setting up a data sharing server.

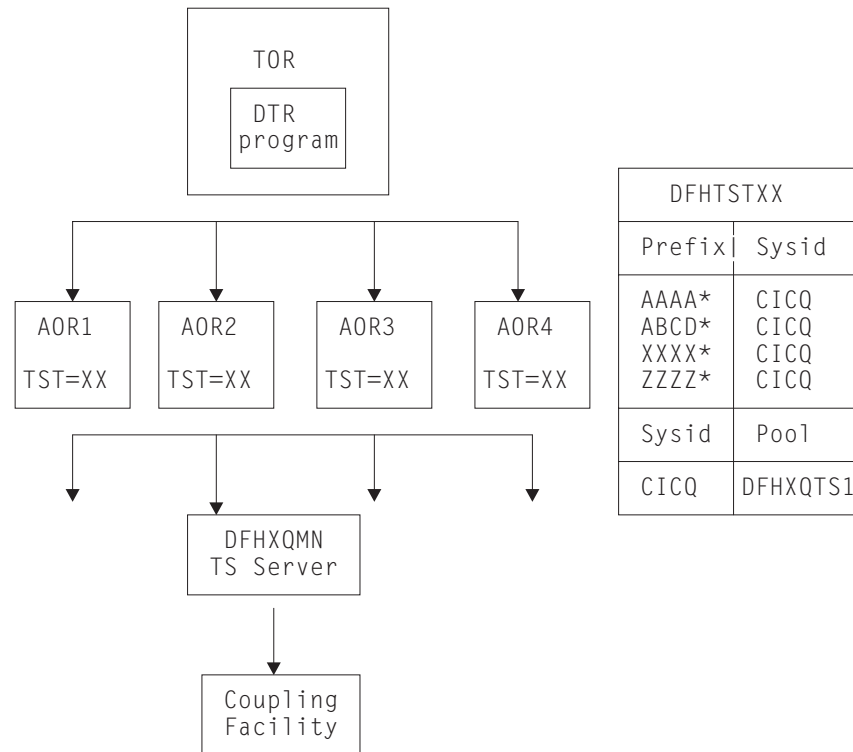


Figure 62. Example of the use of the temporary storage data-sharing server

Naming conventions for remote queues

To define a queue as remote you must include an entry for the queue in a temporary storage table (TST), or use an appropriate TSMODEL. TS queue names are frequently generated dynamically, but they can also be unique fixed names.

- The TST naming convention allows for dynamic names by accepting generic names formed by a constant prefix, to which a CICS application program can add a variable suffix. (Generic names are formed from the leading characters of the 8-character queue names and can be up to seven characters long. Names in a TST entry using all eight characters specify unique TS queues.)
- The names of TS queues defined by TSMODEL resource definitions may have a prefix of up to 16 characters (using a specified set of character) if defined by the Prefix or Remoteprefix option, or of up to 32 characters (using any hexadecimal string) if defined by the XPrefix or XRemoteprefix option. The *CICS Resource Definition Guide* has more information about Prefix, Remoteprefix, XPrefix and XRemoteprefix.

The usual convention is a 4-character prefix (for example, the transaction identifier) followed by a 4-character terminal identifier as the suffix. This generates queue names that are unique for a given terminal. Such generic queue names can be defined easily as remote queues that are owned, for example, by:

- A QOR (thus avoiding transaction affinity problems)
- Shared queues residing in temporary storage data-sharing queue pools
- Remote queues that are owned by a target region, or in a temporary storage data-sharing queue pool

Remote queues and shared queues can be defined in the same way for application programs, but requests for specific SYSIDs are routed to a temporary storage data server by means of TST TYPE=SHARED. However, if the naming convention for dynamically named queues does not conform to this rule, the queue cannot be defined as remote, and all transactions that access the queue must be routed to the target region where the queue was created. Furthermore, a TS queue name cannot be changed from a local to a remote queue name using the global user exits for TS requests.

See Figure 63 for an illustration of the use of a remote queue-owning region.

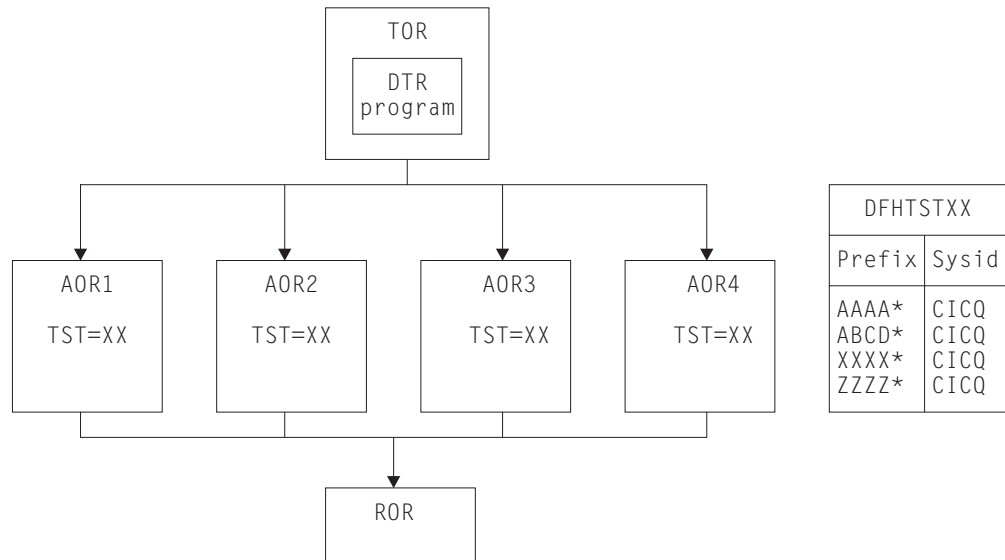


Figure 63. Using remote queues to avoid inter-transaction affinity relating to temporary storage. This example shows a combined file-owning and queue-owning region. Separate regions can be used, but these require special care during recovery operations because of 'in-doubt' windows that can occur when recovering data managed independently by file control and temporary storage control.

Exception conditions for globally accessible queues

When you eliminate inter-transaction affinity relating to TS queues by the use of a global QOR, you must also take care to review exception condition handling. This is because some exception conditions can occur that previously were not possible when the transactions and the queue were local in the same region. This situation arises because the target region and QOR can fail independently, causing circumstances where:

- The queue already exists, because only the target region failed while the QOR continued.
- The queue is not found, because only the QOR failed while the target region continued.

Using transient data

Another form of data queue that CICS application programs commonly use is the transient data queue (TD). The dynamic transaction routing considerations for TD queues have much in common with those for temporary storage. To enable transactions that use a TD queue that needs to be shared, to be dynamically routed to an target region, you must ensure that the TD queues are globally accessible.

All transient data queues must be defined to CICS, and must be installed before the resources become available to a CICS region. These definitions can be changed to support a remote transient data queue-owning region (QOR).

However, there is a restriction for TD queues that use the trigger function. The transaction to be invoked when the trigger level is reached must be defined as a local transaction in the region where the queue resides (in the QOR). Thus the trigger transaction must execute in the QOR. However, any terminal associated with the queue need not be defined as a local terminal in the QOR. This does not create an inter-transaction affinity.

Figure 64 illustrates the use of a remote transient data queue-owning region.

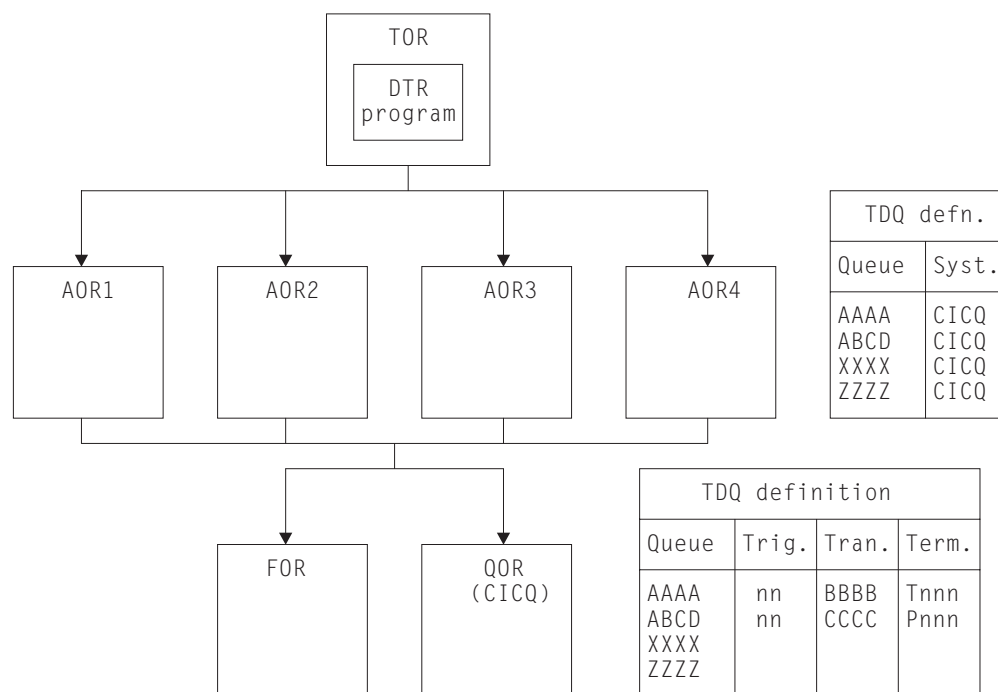


Figure 64. Using remote queues to avoid inter-transaction affinity relating to transient data. The transient data queue definitions installed in the target regions are defined as owned by the QOR (CICQ). All the transient data queue definitions installed in the QOR are local, some with trigger levels.

Exception conditions for globally accessible queues

When you eliminate inter-transaction affinity relating to TD queues by the use of a global QOR, there should not be any new exception conditions (other than SYSIDERR if there is a system definition error or failure).

Using the RETRIEVE WAIT and START commands

The use of some synchronization techniques permit the sharing of task-lifetime storage between two synchronized tasks. For example, the RETRIEVE WAIT and START commands could be used for this purpose, as illustrated in Figure 65.

In this example, TRN1 is designed to retrieve data from an asynchronous task, TRN2, and therefore must wait until TRN2 makes the data available. Note that for this mechanism to work, TRN1 must be a terminal-related transaction.

The steps are as follows:

1. TRN1 writes data to a TS queue, containing its TRANSID and TERMID.
2. To cause itself to suspend, TRN1 issues a RETRIEVE WAIT command, which causes CICS to suspend the task until the RETRIEVE can be satisfied, which is not until TRN2 issues a START command with data passed by the FROM parameter.
3. However, TRN2 can only issue a START command to resume TRN1 if it knows the TRANSID and TERMID of the suspended task (TRN1 in our example). Thus it reads the TS queue to obtain the information written by TRN1. Using a temporary storage queue is one way that this information can be passed by the suspending task.
4. Using the information from the TS queue, TRN2 issues the START command for TRN1, causing CICS to resume TRN1 by satisfying the outstanding RETRIEVE WAIT.

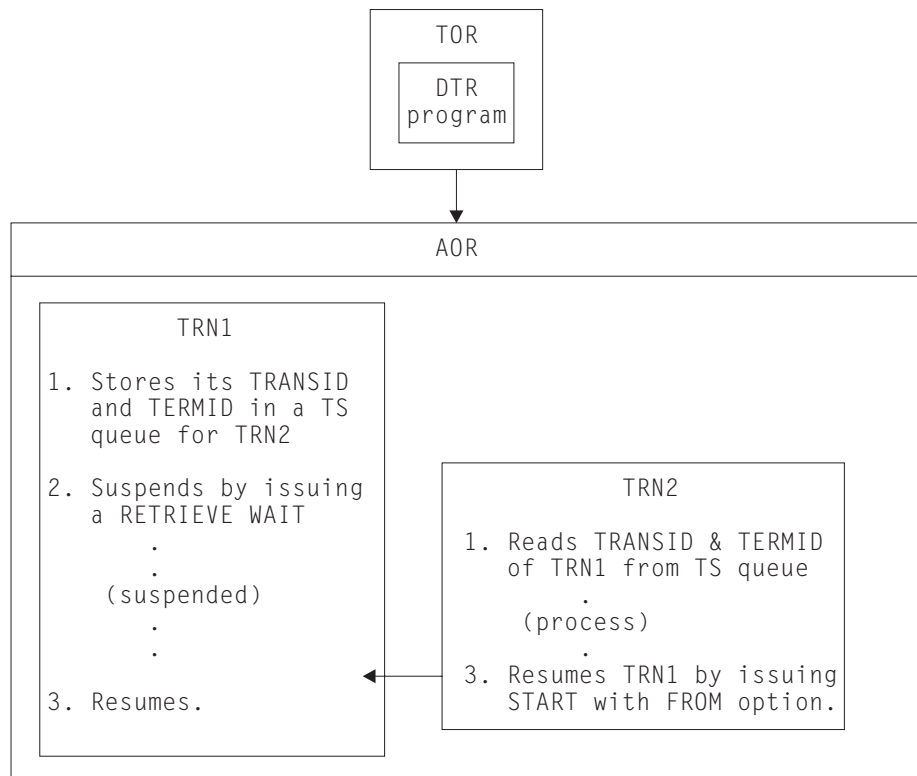


Figure 65. Illustration of task synchronization using RETRIEVE WAIT and START commands

In the example of task synchronization using RETRIEVE WAIT and START commands shown in Figure 65 on page 240, the START command that satisfies the RETRIEVE WAIT must:

- Be issued in same target region as the transaction (TRN1 in our example) issuing the RETRIEVE WAIT command, or
- Specify the SYSID of the target region where the RETRIEVE WAIT command was executed, or
- Specify a TRANSID (TRN1 in our example) that is defined as a remote transaction residing on the target region that executed the RETRIEVE WAIT command.

An application design based on the remote TRANSID technique only works for two target regions. An application design using the SYSID option on the START command only works for multiple target regions if all target regions have connections to all other target regions (which may not be desirable). In either case, the application programs need to be modified: there is no acceptable way to use this programming technique in a dynamic or distributed routing program except by imposing restrictions on the routing program. In general, this means that the dynamic or distributed routing program has to ensure that TRN2 has to execute in the same region as TRN1 to preserve the application design.

Using the START and CANCEL REQID commands

Using this CICS application programming technique, one transaction issues a START command to start another transaction after a specified interval. Another transaction (not the one requested on the START command) determines that it is no longer necessary to run the requested transaction, (which is identified by the REQID parameter) and cancels the START request. Note that the cancel is only effective if the specified interval has not yet expired.

A temporary storage queue is one way that the REQID can be passed from task to task.

Note: To use this technique, the CANCEL command must specify the REQID option, but the START command need not. This is because, provided the NOCHECK option is not specified on the START command, CICS generates a REQID for the request and stores it in the EXEC interface block (EIB) in field EIBREQID.

Figure 66 on page 242 illustrates this programming technique.

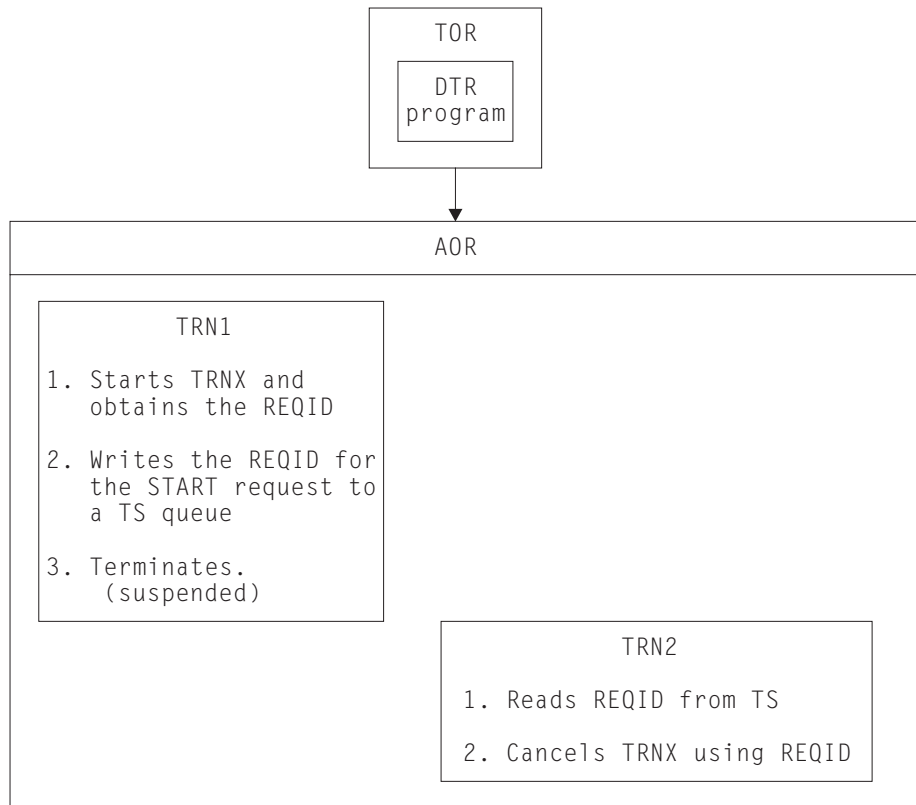


Figure 66. Illustration of the use of the START and CANCEL REQID commands

Using this application programming technique, the CANCEL command that cancels the START request must:

- Be issued in same target region that the START command was executed in, or
- Specify the SYSID of the target region where the START command was executed, or
- Specify a TRANSID (TRNX in our example) that is defined as a remote transaction residing on the target region where the START command was executed.

Note: A START command is not necessarily executed in the same region as the application program issuing the command. It can be function shipped (or, in the case of a non-terminal-related START, routed) and executed in a different CICS region. The above rules apply to the region where the START command is finally executed.

An application design based on the remote TRANSID technique only works for two target regions. An application design using the SYSID option on the cancel command only works for multiple target regions if all target regions have connections to all other target regions. In either case, the application programs need to be modified: there is no acceptable way to use this programming technique in a dynamic or distributed routing program except by imposing restrictions on the routing program.

In general, this means that the dynamic or distributed routing program has to ensure that TRN2 executes in the same region as TRN1 to preserve the application design, and also that TRNX is defined as a local transaction in the same region.

Using the DELAY and CANCEL REQID commands

Using this CICS application programming technique, one task can resume another task that has been suspended by a DELAY command. A DELAY request can only be cancelled by a different task from the one issuing the DELAY command, and the CANCEL command must specify the REQID associated with DELAY command. Both the DELAY and CANCEL command must specify the REQID option to use this technique.

The steps involved in this technique using a temporary storage queue to pass the REQID are as follows:

1. A task (TRN1) writes a predefined DELAY REQID to a TS queue. For example:

```
EXEC CICS WRITEQ TS  
      QUEUE('DELAYQUE') FROM(reqid_value)
```

2. The task waits on another task by issuing a DELAY command, using the *reqid_value* as the REQID. For example:

```
EXEC CICS DELAY  
      INTERVAL(1000) REQID(reqid_value)
```

3. Another task, TRN2, reads the REQID of the DELAY request from TS queue called 'DELAYQUE'.

4. TRN2 completes its processing, and resumes TRN1 by cancelling the DELAY request.

The process using a TS queue is illustrated in Figure 67.

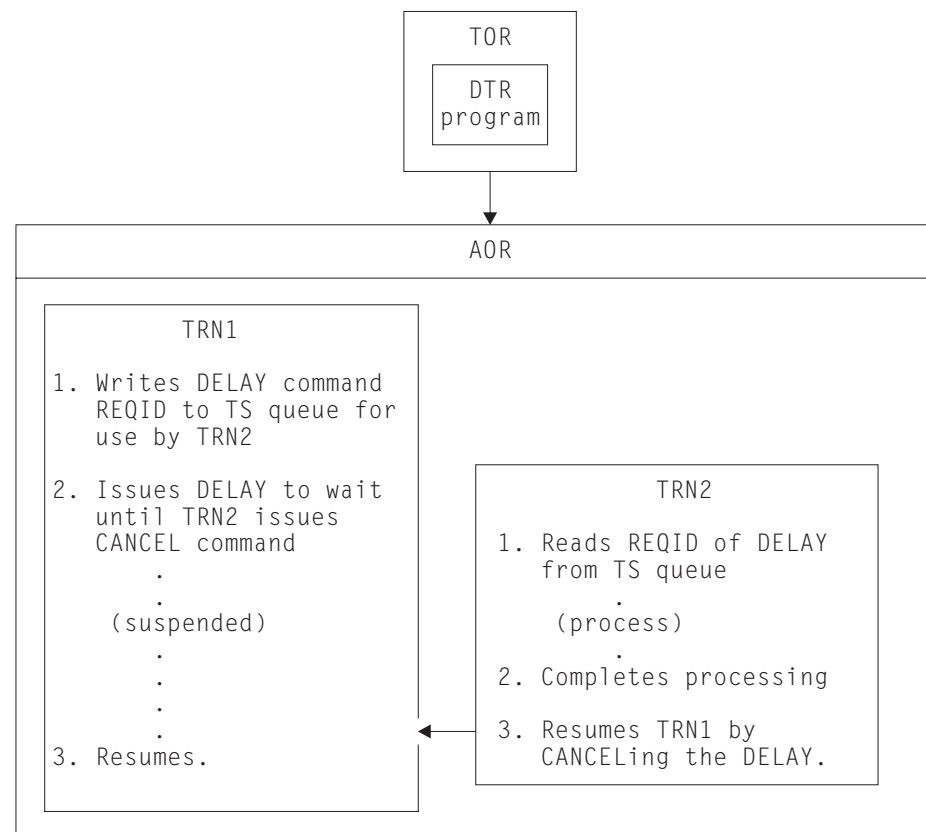


Figure 67. Illustration of the use of the DELAY and CANCEL REQID commands

Another way to pass the REQID when employing this technique would be for TRN1 to start TRN2 using the START command with the FROM and TERMID options. TRN2 could then obtain the REQID with the RETRIEVE command, using the INTO option.

Using this application programming technique, the CANCEL command that cancels the DELAY request must:

- Be issued in same target region as the DELAY command was executed in, or
- Specify the SYSID of the target region where the DELAY command was executed, or
- Specify a TRANSID (TRN1 in our example) that is defined as a remote transaction residing on the target region where the DELAY command was executed.

An application design based on the remote TRANSID technique only works for two target regions. An application design using the SYSID option on the cancel command only works for multiple target regions if all target regions have connections to all other target regions. In either case, the application programs need to be modified: there is no acceptable way to use this programming technique in a dynamic or distributed routing environment except by imposing restrictions on the routing program.

If the CANCEL command is issued by a transaction that is initiated from a terminal, it is possible that the transaction could be dynamically routed to the wrong target region.

Using the POST and CANCEL REQID commands

The CICS POST command is used to request notification that a specified time has expired. Another transaction (TRN2) can force notification, as if the specified time has expired, by issuing a CANCEL of the POST request.

The time limit is signalled (posted) by CICS by setting a bit pattern in the event control block (ECB). To determine whether notification has been received, the requesting transaction (TRN1) has either to test the ECB periodically, or to issue a WAIT command on the ECB.

A TS storage queue is one way that can be used to pass the REQID of the POST request from task to task.

Figure 68 on page 245 illustrates this technique.

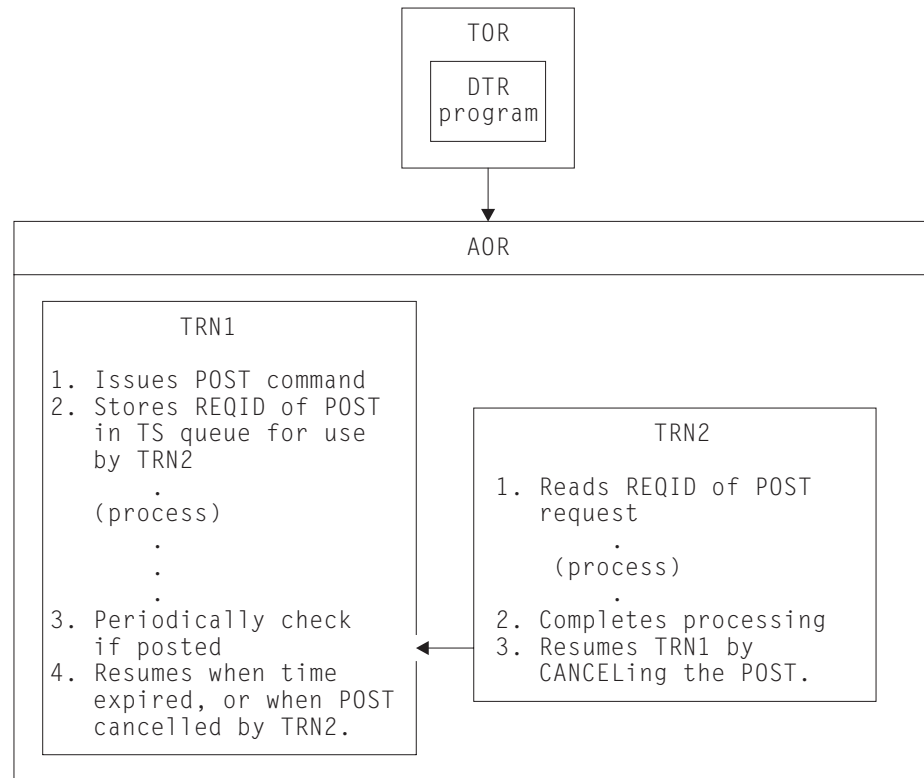


Figure 68. Illustration of the use of the POST command

If this technique is used, the dynamic or distributed routing program has to ensure that TRN2 executes in the same CICS region as TRN1 to preserve the application design.

The CANCEL command that notifies the task that issued the POST must:

- Be issued in same target region that the POST command was executed in, or
- Specify the SYSID of the target region where the POST command was executed, or
- Specify a TRANSID (TRN1 in our example) that is defined as a remote transaction residing on the target region where the POST command was executed.

An application design based on the remote TRANSID technique only works for two target regions. An application design using the SYSID option on the cancel command only works for multiple target regions if all target regions have connections to all other target regions. In either case, the application programs need to be modified: there is no acceptable way to use this programming technique in a dynamic or distributed routing program except by imposing restrictions on the routing program.

In general, this means that the dynamic or distributed routing program has to ensure that TRN2 executes in the same region as TRN1 to preserve the application design.

Clearly, there is no problem if the CANCEL is issued by the same task that issued the POST. If a different task cancels the POST command, it must specify REQID to identify the particular instance of that command. Hence the CANCEL command with

REQID is indicative of an inter-transaction affinity problem. However, REQID need not be specified on the POST command because CICS automatically generates a REQID and pass it to the application in EIBREQID.

Detecting inter-transaction affinities

To manage transaction affinities in a dynamic routing environment, you must first discover which transactions have affinities. How do you do this?

The recommended way is to use the CICS Interdependency Analyzer to detect affinities. The *CICS Interdependency Analyzer for z/OS User's Guide and Reference* describes the utility and how to use it.

Note: If you dynamically route program-link requests, you must discover which programs (or their associated transactions) have affinities. You cannot use the CICS Interdependency Analyzer to do this.

If you do not use the utility, you can use one of the following methods to detect affinities, although you are strongly recommended to use the utility.

- Review application design, paying particular attention to the techniques used for inter-transaction communication.
- Search the source of application programs, looking for instances of the EXEC CICS commands that can give rise to inter-transaction affinity.
- Run a trace analysis program that can analyze CICS auxiliary trace. For example, if you run the CICS trace utility program, DFHTUP, with the ABBREV option to format CICS auxiliary trace output, you can analyze the resulting abbreviated trace data to find instances of suspect commands.

Inter-transaction affinities caused by application generators

Application generators may give rise to particularly difficult problems of inter-transaction affinity:

- The affinity may be hidden from the application programmer.
- The application generator may have a different concept of a transaction to CICS: it is affinity among CICS transactions that is of concern, because these are the entities that are dynamically routed.
- Some application generators use a single transaction code for all transactions within an application, making it difficult for the router to select those instances of transactions that have affinities.

Duration and scope of inter-transaction affinities

When planning your dynamic routing strategy, and planning how to manage inter-transaction affinities, it is important to understand the concepts of affinity relations and affinity lifetimes. The relations and lifetimes of inter-transaction affinities must be taken into account when designing a dynamic or distributed routing program, because they define the scope and duration of inter-transaction affinities. Clearly, the ideal situation for a dynamic or distributed routing program is for there to be no inter-transaction affinities at all, which means there are no restrictions in the choice of available target regions for dynamic routing. However, even when inter-transaction affinities do exist, there are limits to the scope of these affinities, the scope of the affinity being determined by affinity relation and affinity lifetime.

Understanding the relations and lifetimes of transaction affinities is important in deciding how to manage them in a dynamic routing environment.

Affinity transaction groups

In order to manage affinities within a dynamic routing environment, you must first categorize transactions by their affinity. One way to do this is to place transactions in groups, where a group is a set of transactions that have inter-transaction affinity. Each affinity transaction group (or affinity group, for short) thus represents a group of transactions that have an affinity with one another. Defining affinity groups is one way that a dynamic or distributed routing program can determine to which target region a transaction should be routed.

Clearly, the more inter-transaction affinity you have in a given CICS workload, the less effective a dynamic routing program can be in balancing the workload across a CICSplex. To minimize the impact of inter-transaction affinity, affinities within an affinity group can be characterized by their relation and lifetime. These relation and lifetime attributes determine the scope and duration of an affinity.

Thus, ideally, an affinity transaction group consists of an affinity group identifier, a set of transactions that constitute the affinity group, with the affinity relation and affinity lifetime associated with the group.

Relations and lifetimes

When you create an affinity group, you should assign to the group the appropriate affinity relation and affinity lifetime attributes. The relation determines how the dynamic or distributed routing program is to select a target region for a transaction instance associated with the affinity, and the lifetime determines when the affinity is ended.

There are four possible affinity relations that you can assign to your affinity groups:

1. Global
2. LUsername
3. Userid
4. BAPPL

These are described in the following sections, together with the permitted lifetimes for each relation.

The global relation

A group of transactions whose affinity relation is defined as global is one where *all* instances of all transactions in the group that are initiated from any terminal, by any START command, or by any CICS BTS process, must execute in the same target region for the lifetime of the affinity. The affinity lifetime for global relations can be as follows:

System

The affinity lasts for as long as the target region exists, and ends whenever the target region terminates (at a normal, immediate, or abnormal termination).

Permanent

The affinity extends across all CICS restarts. This is the most restrictive of all the inter-transaction affinities. If you are running CICSplex SM, this affinity lasts for as long as any CMAS involved in managing the CICSplex using the workload is active.

An example of a global inter-transaction affinity with a lifetime of permanent is where the transaction uses (reads and/or writes) a local, recoverable, temporary storage queue, and where the TS queue name is **not** derived from the terminal. (You can only specify that a TS queue is recoverable in the CICS region in which the queue is local.)

Generally, transactions in this affinity category are not suitable candidates for dynamic routing and you should consider making them statically routed transactions.

An example of a global relation is illustrated in Figure 69.

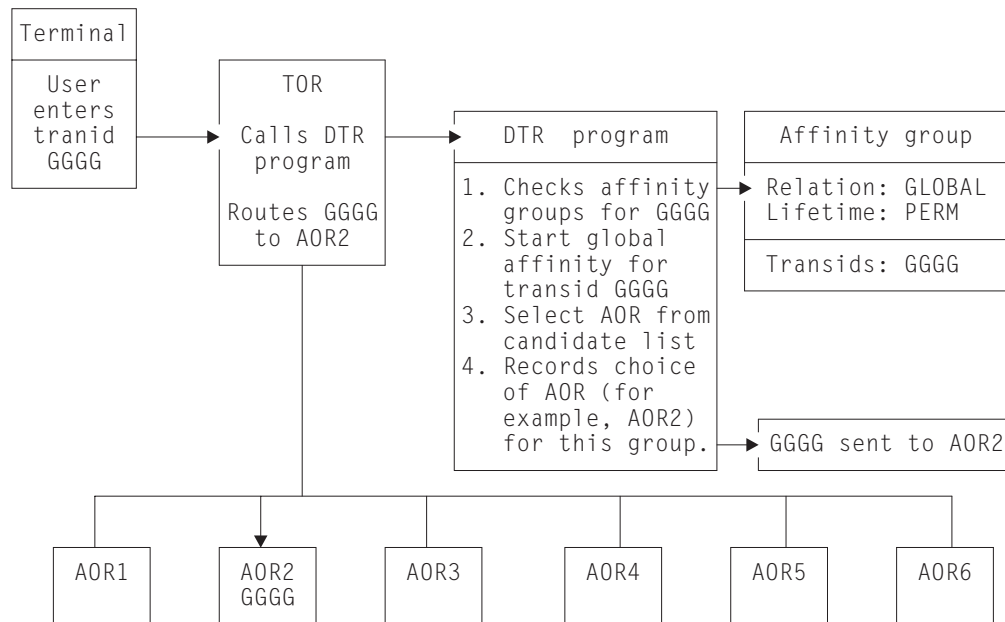


Figure 69. Managing inter-transaction affinity with global relation and permanent lifetime

In this example, the transaction GGGG is defined in a group with a permanent global affinity relation. The first instance of transid GGGG, from any terminal, starts a permanent-lifetime affinity. The first instance of GGGG can be routed to any suitable target region. In this example, AOR2 is selected from the possible range AOR1 through AOR6, but all other instances, from any terminal, must also be routed to the same region, AOR2.

The LUname (terminal) relation

A group of transactions whose affinity relation is defined as LUname is one where **all** instances of all transactions in the group that are associated with the **same terminal** must execute in the same target region for the lifetime of the affinity. The affinity lifetime for LUname relations can be as follows:

Pseudoconversation

The affinity lasts for the whole pseudoconversation, and ends when the pseudoconversation ends at the terminal. Each transaction must end with an EXEC CICS RETURN TRANSID, not with the pseudoconversation mode of END.

Logon

The affinity lasts for as long as the terminal remains logged-on to CICS, and ends when the terminal logs off.

System

The affinity lasts for as long as the target region exists, and ends whenever the target region terminates (at a normal, immediate, or abnormal termination).

Permanent

The affinity extends across all CICS restarts. If you are running CICSplex SM, this affinity lasts for as long as any CMAS involved in managing the CICSplex using the workload is active.

Delimit

The affinity continues until a transaction with a pseudoconversation mode of END is encountered.

A typical example of transactions that have an LUnicode relation are those that:

- Use a local TS queue to pass data between the transactions in a pseudoconversation, and
- The TS queue name is derived, in part, from the terminal name (see “Naming conventions for remote queues” on page 237 for information about TS queue names).

These types of transaction can be placed in an affinity group with a relation of terminal and lifetime of pseudoconversation. When the dynamic routing program detects the first transaction in the pseudoconversation initiated by a specific terminal (LUnicode), it is free to route the transaction to any target region that is a valid candidate for that transaction. However, any subsequent transaction within the affinity group that is initiated at **the same terminal** must be routed to the same target region as the transaction that started the pseudoconversation. When the affinity ends (at the end of the pseudoconversation on a given terminal), the dynamic routing program is again free to route the first transaction to any candidate target region.

This form of affinity is manageable and does not impose too severe a constraint on dynamic transaction routing, and may occur commonly in many CICSplexes. It can be managed easily by a dynamic routing program, and should not inhibit the use of dynamic routing.

This example is illustrated in Figure 70 on page 250.

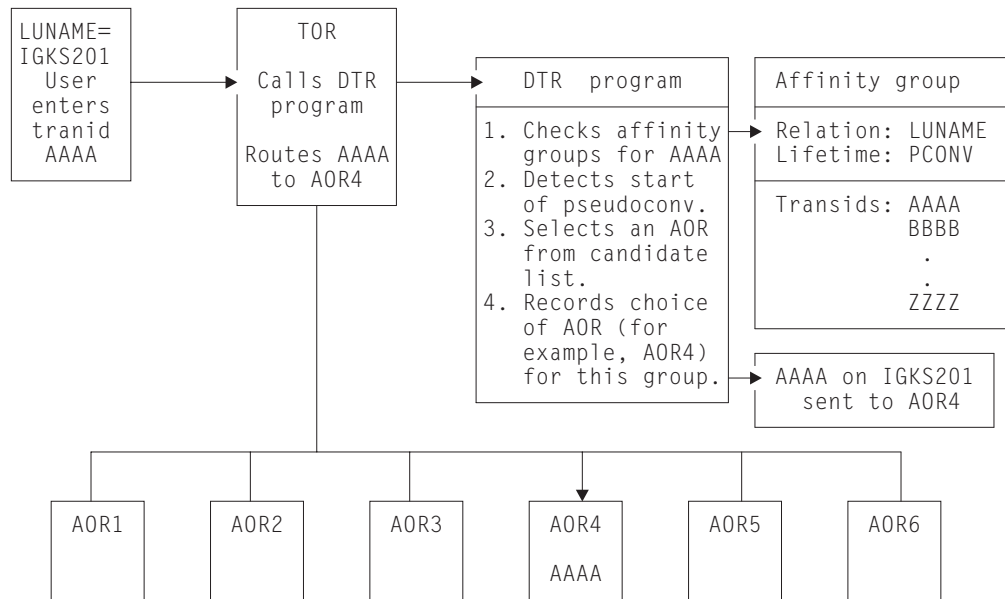


Figure 70. Managing inter-transaction affinity with LUNAME relation and pseudoconversation lifetime

In this example, each instance of transid AAAA from a terminal starts a pseudoconversational-lifetime affinity. AAAA can be routed to any suitable target region (AOR1 through AOR6), but other transactions in the group, in the same pseudoconversation with the same terminal (IGKS201 in this example) must be routed to whichever target region is selected for AAAA.

The userid relation

A group of transactions whose affinity relation is defined as userid is one where **all** instances of the transactions that are initiated from a terminal, by a START command, or by a CICS BTS activity, and executed on behalf of the **same userid**, must execute in the same target region for the lifetime of the affinity. The affinity lifetime for userid relations can be as follows:

Pseudoconversation

The affinity lasts for the whole pseudoconversation, and ends when the pseudoconversation ends for that userid. Each transaction must end with an EXEC CICS RETURN TRANSID, not with the pseudoconversation mode of END.

Signon

The affinity lasts for as long as the user is signed on, and ends when the user signs off. Note this lifetime is only possible in those situations where only one user per userid is permitted. Signon lifetime cannot be detected if multiple users are permitted to be signed on with the same userid at the same time (at different terminals).

System

The affinity lasts for as long as the target region exists, and ends whenever the target region terminates (at a normal, immediate, or abnormal termination).

Permanent

The affinity extends across all CICS restarts. If you are running CICSplex SM, this affinity lasts for as long as any CMAS involved in managing the CICSplex using the workload is active.

Delimit

The affinity continues until a transaction with a pseudoconversation mode of END is encountered.

A typical example of transactions that have a userid relation is where the userid is used dynamically to identify a resource, such as a TS queue. The least restrictive of the affinities in this category is one that lasts only for as long as the user remains signed on.

An example of an affinity group with the userid relation and a signon lifetime is shown in Figure 71.

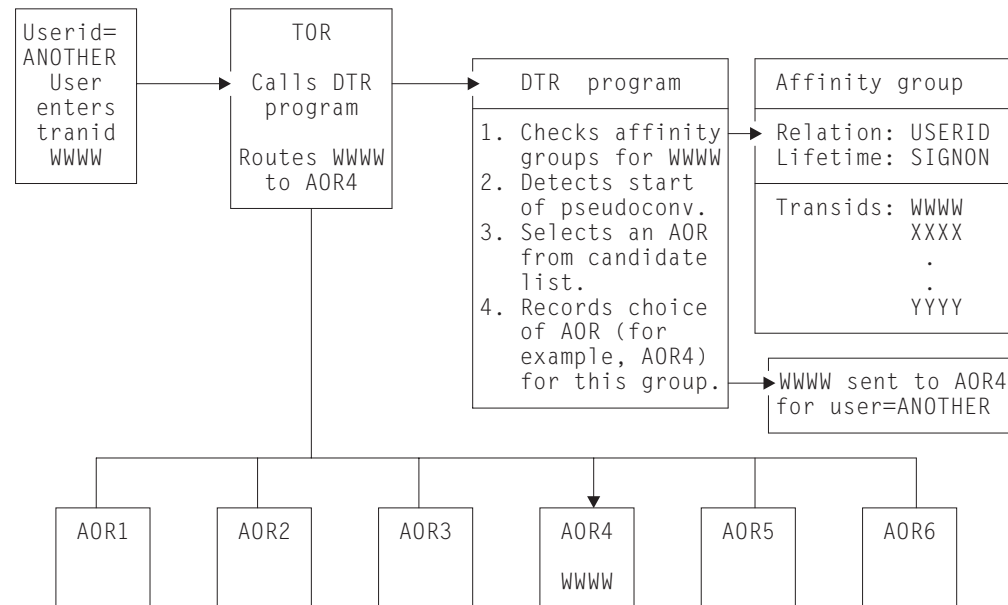


Figure 71. Managing inter-transaction affinity with userid relation and sign-on lifetime

In this example, any instance of a transaction from a terminal starts a sign-on lifetime affinity. It can be routed to any suitable target region (AOR1 through AOR6), but other transactions in the group for the same user (ANOTHER in this example) must be routed to whichever target region is selected for the first instance of a transaction in the group.

The BAPPL relation

A group of transactions whose affinity relation is defined as BAPPL is one where **all** instances of all transactions in the group that are associated with the same BTS process are to be directed to the same target region. The affinity lifetimes for BAPPL relations can be as follows:

Process

The affinity lasts for as long as the associated process exists.

Activity

The affinity lasts for as long as the associated activity exists.

System

The affinity lasts for as long as the target region exists, and ends whenever the target region terminates (at a normal, immediate, or abnormal termination).

Permanent

The affinity extends across all CICS restarts. If you are running CICSplex SM, this affinity lasts for as long as any CMAS involved in managing the CICSplex using the workload is active.

A typical example of transactions that have a BAPPL relation is where a local temporary storage queue is used to pass data between the transactions within a BTS activity or process.

An example of an affinity group with the BAPPL relation is shown in Figure 72.

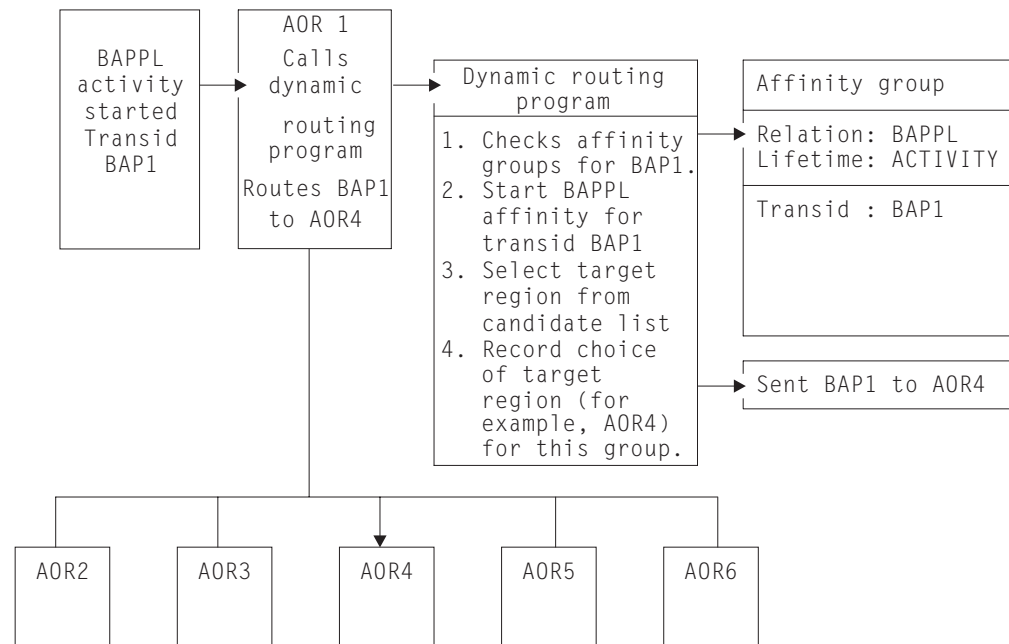


Figure 72. Managing inter-transaction affinity with BAPPL relation and activity lifetime

In this example, the first instance of BTS transaction BAP1 starts a BAPPL–activity affinity. The first instance of BAP1 can be routed to any suitable target region (AOR1 through AOR6), but all other instances of the activity must be routed to whichever target region is selected for BAP1.

Although BTS itself does not introduce any affinities, and discourages programming techniques that do, it does support existing code that may introduce affinities. You must define such affinities to workload management. It is particularly important to specify each affinity's lifetime. Failure to do this may restrict unnecessarily the workload management routing options.

It is important to note that a given activity can be run both synchronously and asynchronously. Workload management is only able to honour invocations that are made asynchronously. Furthermore, you are strongly encouraged not to create these affinities, particularly activity and process affinities, because these affinities are synchronized across the BTS-set. This could have serious performance impacts on your systems.

You should also note that, with CICSplex SM, the longest time that an affinity can be maintained is while a CMAS involved in the workload is active; that is, an affinity

of PERMANENT. If there is a total system failure, or a planned shutdown, affinities will be lost, but activities in CICS will be recovered from the BTS RLS data set.

Chapter 16. Recovery design

CICS provides two techniques that can help you to recover or reconstruct events or data changes during CICS execution:

- “Journaling”
- “Syncpointing” on page 257

Techniques for named counter recovery are described in “Named counter recovery” on page 358.

Journaling

CICS provides facilities for creating and managing **journals** during CICS processing. Journals may contain any and all data the user needs to facilitate subsequent reconstruction of events or data changes. For example, a journal might act as an audit trail, a change-file of database updates and additions, or a record of transactions passing through the system (often referred to as a **log**). Each journal can be written from any task.

Journal control commands are provided to allow the application programmer to:

- Create a journal record (WRITE JOURNALNAME or WRITE JOURNALNUM command)
- Synchronize with (wait for completion of) journal output (WAIT JOURNALNAME or WAIT JOURNALNUM command)

Exception conditions that occur during execution of a journal control command are handled as described in Chapter 17, “Dealing with exception conditions,” on page 261. (The earlier JFILEID option is supported for compatibility purposes only.)

Journal records

Each journal is identified by a name or number known as the journal identifier. This number may range from 1 through 99. The name DFHLOG is reserved for the journal known as the system log.

When a journal record is built, the data is moved to the journal buffer area. All buffer space and other work areas needed for journal operations are acquired and managed by CICS. The user task supplies only the data to be written to the journal. Log manager is designed so that the application programmer requesting output services does not have to be concerned with the detailed layout and precise contents of journal records. The programmer has to know only which journal to use, what user data to specify, and which user-identifier to supply.

Journal output synchronization

When a synchronous journal record is created by issuing the WRITE JOURNALNAME or WRITE JOURNALNUM command with the WAIT option, the requesting task can wait until the output has been completed. By specifying that this should happen, the application programmer ensures that the journal record is written on the external storage device associated with the journal before processing continues; the task is said to be **synchronized** with the output operation.

The application programmer can also request asynchronous journal output. This causes a journal record to be created in the journal buffer area but allows the requesting task to retain control and thus to continue with other processing. The

task may check and wait for output completion (that is, synchronize) later by issuing the WAIT JOURNALNAME or WAIT JOURNALNUM command.

Note: In some cases, a SHUTDOWN IMMEDIATE can cause user journal records to be lost, if they have been written to a log manager buffer but not to external storage. This is also the case if the CICS shut-down assist transaction (CESD) forces SHUTDOWN IMMEDIATE during a normal shutdown, because normal shutdown is hanging. To avoid the risk of losing journal records, you are recommended to issue CICS WAIT JOURNALNUM requests periodically, and before ending your program.

Without WAIT, CICS does not write data to the log stream until it has a full buffer of data, or until some other unrelated activity requests that the buffer be hardened, thus reducing the number of I/O operations. Using WAIT makes it more difficult for CICS to calculate accurately log structure buffer sizes. For CF log streams, this could lead to inefficient use of storage in the coupling facility.

The basic process of building journal records in the CICS buffer space of a given journal continues until one of the following events occurs:

- For system logs:
 - Whenever the system requires it to ensure integrity and to permit a future emergency restart
 - The log stream buffer is filled
- For user journals:
 - The log stream buffer is filled (or, if the journal resides on SMF, when the journal buffer is filled)
 - A request specifying the WAIT option is made (from any task) for output of a journal record
 - An EXEC CICS SET JOURNALNAME command is issued
 - An EXEC CICS DISCARD JOURNALNAME command is issued
 - Any of the above occurring for any other journal which maps onto the same log stream
 - On a normal shutdown
- For forward recovery logs:
 - The log stream buffer is filled
 - At syncpoint (first phase)
 - On file closure
- For autojournals:
 - The log stream buffer is filled
 - A request specifying the WAIT option is made (from any task) for output of a journal record
 - On file closure
- For the log-of-logs (DFHLGLOG):
 - On file OPEN and CLOSE requests

When any one of these occurs, all journal records present in the buffer, including any deferred output resulting from asynchronous requests, are written to the log stream as one block.

The advantages that may be gained by deferring journal output are:

- Transactions may get better response times by waiting less.

- The load of physical I/O requests on the host system may be reduced.
- Log streams may contain fewer but larger blocks and so better utilize primary storage.

However, these advantages are achievable only at the cost of greater programming complexity. It is necessary to plan and program to control synchronizing with journal output. Additional decisions that depend on the data content of the journal record and how it is to be used must be made in the application program. In any case, the full benefit of deferring journal output is obtained only when the load on the journal is high.

If the journal buffer space available at the time of the request is not sufficient to contain the journal record, the NOJBUFSP condition occurs. If no HANDLE CONDITION command is active for this condition, the requesting task loses control, the contents of the current buffer are written, and the journal record is built in the resulting freed buffer space before control returns to the requesting task.

If the requesting task is not willing to lose control (for example, if some housekeeping must be performed before other tasks get control), a HANDLE CONDITION command should be issued. If the NOJBUFSP condition occurs, no journal record is built for the request, and control is returned directly to the requesting program at the location provided in the HANDLE CONDITION command. The requesting program can perform any housekeeping needed before reissuing the journal output request.

Journal commands can cause immediate or deferred output to the journal. System log records are distinguished from all other records by specifying JOURNALNAME(DFHLOG) on the request. User journal records are created using some other JOURNALNAME or a JOURNALNUM. All records must include a journal type identifier, (JTYPEID). If the user journaling is to the system log, the journal type identifier (according to the setting of the high-order bit) also serves to control the presentation of these to the global user exit XRCINPT at a warm or emergency restart. Records are presented during the backward scan of the log as follows:

- For in-flight or in-doubt tasks only (high-order bit off)
- For all records encountered until the scan is terminated (high-order bit on)

See the *CICS Customization Guide* for information about the format and structure of journal records. See the section on emergency restart in the *CICS Recovery and Restart Guide* for background information and a description of the recovery process.

Syncpointing

To facilitate recovery in the event of abnormal termination of a CICS task or of failure of the CICS system, the system programmer can, during CICS table generation, define specific resources (for example, files) as recoverable. If a task is terminated abnormally, these resources are restored to the condition they were in at the start of the task, and can then be rerun. The process of restoring the resources associated with a task is termed **backout**.

If an individual task fails, backout is performed by the dynamic transaction backout program. If the CICS system fails, backout is performed as part of the emergency restart process. See the *CICS Recovery and Restart Guide* which describes these facilities, which in general have no effect on the coding of application programs.

However, for long-running programs, it may be undesirable to have a large number of changes, accumulated over a period of time, exposed to the possibility of backout in the event of task or system failure. This possibility can be avoided by using the SYNCPOINT command to split the program into logically separate sections known as units of work (UOWs); the end of an UOW is referred to as a synchronization point (**syncpoint**). For more information about syncpoints, see the *CICS Recovery and Restart Guide*.

If failure occurs after a syncpoint but before the task has been completed, only changes made after the syncpoint are backed out.

Alternatively, you can use the SAA Resource Recovery interface instead of the SYNCPOINT command. This provides an alternative API to existing CICS resource recovery services. You may wish to use the SAA Resource Recovery interface in networks that include multiple SAA platforms, where the consistency of a common API is seen to be of benefit. In a CICS system, the SAA Resource Recovery interface provides the same function as the EXEC CICS API.²

The SAA Resource Recovery interface is implemented as a call interface, having two call types:

SRRCMIT

Commit—Equivalent to SYNCPOINT command.

SRRBACK

Backout—Equivalent to SYNCPOINT ROLLBACK command.

For further information about the SAA Resource Recovery interface, see *SAA Common Programming Interface for Resource Recovery Reference* manual.

UOWs should be entirely logically independent, not merely with regard to protected resources, but also with regard to execution flow. Typically, an UOW comprises a complete conversational operation bounded by SEND and RECEIVE commands. A browse is another example of an UOW; an ENDBR command must therefore precede the syncpoint.

In addition to a DL/I termination call being considered to be a syncpoint, the execution of a SYNCPOINT command causes CICS to issue a DL/I termination call. If a DL/I PSB is required in a subsequent UOW, it must be rescheduled using a program control block (PCB) call or a SCHEDULE command.

With distributed program link (DPL), it is possible to specify that a syncpoint is taken in the server program, to commit the server resources before returning control to the client. This is achieved by using the SYNCONRETURN option on the LINK command. For programming information about the SYNCONRETURN option, see "The SYNCONRETURN option for the server program" on page 5 on page 440 and the *CICS Application Programming Reference* manual.

A BMS logical message, started but not completed when a SYNCPOINT command is processed, is forced to completion by an implied SEND PAGE command. However, you should not rely on this because a logical message whose first page is incomplete is lost. You should also code an explicit SEND PAGE command before the SYNCPOINT command or before termination of the transaction.

2. Full SAA Resource Recovery provides some return codes that are not supported in its CICS implementation. (See the CICS appendix in the *SAA Common Programming Interface for Resource Recovery Reference* manual.)

Consult your system programmer if syncpoints are to be issued in a transaction that is eligible for transaction restart.

Chapter 17. Dealing with exception conditions

Every time you process an EXEC CICS command in one of your applications, CICS automatically raises a condition, or return code, to tell you what happened. You can choose to have this condition, which is usually NORMAL, passed back by the CICS EXEC interface program to your application. It is sometimes called a RESP value, because you may get hold of it by using the RESP option in your command. Alternatively, you may obtain this value by reading it from the EXEC interface block (EIB).

If something out of the ordinary happens, you get an *exception condition*, which simply means a condition other than NORMAL. By testing this condition, you can find out what has happened and, possibly, why.

Many exception conditions have an additional (RESP2) value associated with them, which gives further information. You may obtain this RESP2 value either by using the RESP2 option in your command in addition to the RESP option, or by reading it from the EIB.

Not all conditions denote an error situation, even if they are not NORMAL. For example, if you get an ENDFILE condition on a READNEXT command during a file browse, it might be exactly what you expect. For information about all possible conditions and the commands on which they can occur, see the *CICS Application Programming Reference* manual.

This chapter describes:

- Default CICS exception handling
- “Handling exception conditions by in-line code” on page 262
- “Modifying default CICS exception handling” on page 265

Default CICS exception handling

If your application is written in a language other than C, C++, or Java and you do not specify otherwise, CICS uses its built-in exception handling whenever an exception condition occurs. If your application is written in C or C++, CICS itself takes no action when an exception condition occurs and it is left to the application to handle it. See “Handling exception conditions by in-line code” on page 262 for information on handling exception conditions.

The most common action by CICS is to cause an abend of some type to happen. The particular behaviors for each condition and for each command are detailed in the *CICS Application Programming Reference* and *CICS System Programming Reference* manuals.

Sometimes you will be satisfied with the CICS default exception handling, in which case you need do nothing. More often you will prefer some other course of action.

These are the different ways of turning off the default CICS handling of exception conditions.

- Turn off the default CICS handling of exception conditions on a particular EXEC CICS command call by specifying the NOHANDLE option.
- Alternatively, turn off the default CICS handling of exception conditions by specifying the RESP option on the command. This, of itself, switches off the default CICS exception handling in the same way as NOHANDLE does. It also

causes the variable named by the argument of RESP to be updated with the value of the condition returned by the command. This is described in more detail in “Handling exception conditions by in-line code.”

- Write your application program in C or C++.

If the default CICS exception handling is turned off you should ensure that your program copes with anything that may happen in the command call.

The traditional, but no longer recommended, way to specify some other course of action is available only if you are programming in a language other than C or C++: it is to use combinations of the HANDLE ABEND, HANDLE CONDITION, and IGNORE CONDITION commands to modify the default CICS exception handling. This is described in “Modifying default CICS exception handling” on page 265.

Handling exception conditions by in-line code

This section describes the method of handling exception conditions which is recommended for new applications and is the only available choice if your programs are in C or C++ language. If your program is not written in C or C++, it involves either using the NOHANDLE option or specifying the RESP option on EXEC CICS commands, which prevents CICS performing its default exception handling. Additionally, the RESP option makes the value of the exception condition directly available to your program, for it to take remedial action.

If your program is written in C or C++, in-line code is the only means you have of handling exception conditions.

If you use the NOHANDLE or RESP option, you should ensure that your program can cope with whatever condition may arise in the course of executing the commands. The RESP value is available to enable your program to decide what to do and more information which it may need to use is carried in the EXEC interface block (EIB). In particular, the RESP2 value is contained in one of the fields of the EIB. See the *CICS Application Programming Reference* manual for more information on the EIB. Alternatively, if your program specifies RESP2 in the command, the RESP2 value is returned by CICS directly.

The DFHRESP built-in translator function makes it very easy to test the RESP value. It allows you to examine RESP values symbolically. This is easier than examining binary values that are less meaningful to someone reading the code.

How to use the RESP and RESP2 options

The argument of RESP is a user-defined fullword binary data area (long integer). On return from the command, it contains a value corresponding to the condition that may have been raised. Normally its value is DFHRESP(NORMAL).

Use of RESP and DFHRESP in COBOL and PL/I

Here is an example of an EXEC CICS call in COBOL which uses the RESP option. A PL/I example would be similar, but would end in “;” instead of END-EXEC.

```
EXEC CICS WRITEQ TS FROM(abc)
      QUEUE(qname)
      NOSUSPEND
      RESP(xxx)
END-EXEC.
```

An example of using DFHRESP to check the RESP value is:

```
IF xxx=DFHRESP(NOSPACE) THEN ...
```

Use of RESP and DFHRESP in C and C++

Here is an example of an EXEC CICS call in C, which uses the RESP option, including the declaration of the RESP variable:

```
long response;
...
EXEC CICS WRITEQ TS FROM(abc)
        QUEUE(qname)
        NOSUSPEND
        RESP(response);
```

An example of using DFHRESP to check the RESP value is:

```
if (response == DFHRESP(NOSPACE))
{
...
}
```

Use of DFHRESP in assembler

An example of a test for the RESP value in assembler language is:

```
CLC    xxx,DFHRESP(NOSPACE)
BE     ...
```

An example of exception handling in C

The following example is a typical function which could be used to receive a BMS map and to cope with exception conditions:

```
int ReadAccountMap(char *mapname, void *map)
{
    long    response;
    int     ExitKey;
    EXEC CICS RECEIVE MAP(mapname)
            MAPSET("ACCOUNT")
            INTO(map)
            RESP(response);
    switch (response)
    {
    case DFHRESP(NORMAL):
        ExitKey = dfheiptr->eibaid;
        ModifyMap(map);
        break;
    case DFHRESP(MAPFAIL):
        ExitKey = dfheiptr->eibaid;
        break;
    default:
        ExitKey = DFHCLEAR;
        break;
    }
    return ExitKey;
}
```

Figure 73. An example of exception handling in C

The *ReadAccountMap* function has two arguments:

1. *mapname* is the variable which contains the name of the map which is to be received.
2. *map* is the address of the area in memory to which the map is to be written.

The RESP value will be returned in *response*. The declaration of *response* sets up the appropriate type of automatic variable.

The condition handling can be done by using if statements. However, to improve readability, it is often better, as here, to use a switch statement, instead of compound if ... else statements. The effect on program execution time is negligible.

1. A condition of NORMAL is what is normally expected. If a condition of NORMAL is detected in the example here, the function then finds out what key the user pressed to return to CICS and this value is passed to ExitKey. The program then makes some update to the map held in memory by the ModifyMap function, which need not concern us further.
2. A condition of MAPFAIL, signifying that the user has made no updates to the screen, is also fairly normal and is specifically dealt with here. In this case the program again updates ExitKey but does not call ModifyMap.

An example of exception handling in COBOL

```
#
#      03      RESPONSE                                PIC S9(8)   BINARY.
#      03      EXITKEY                                PIC X.
#      .
#
# EXEC CICS RECEIVE MAP(MAPNAME)
#              MAPSET('ACCOUNT')
#              INTO(MAP)
#              RESP(RESPONSE)
#              END-EXEC.
#
# IF (RESPONSE NOT = DFHRESP(NORMAL)) AND
#    (RESPONSE NOT = DFHRESP(MAPFAIL))
#     MOVE DFHCLEAR TO EXITKEY
# ELSE
#     MOVE EIBAID TO EXITKEY
#     IF RESPONSE = DFHRESP(NORMAL)
#         GO TO MODIFYMAP
#     END-IF
# END-IF.
#
#
#
#
#
#
#
#
#
#
```

Figure 74. An example of exception handling in COBOL

The RESP value is returned in RESPONSE. RESPONSE is declared as a fullword binary variable in the data section.

The EXEC CICS statement asks for a map of the name given by *MAPNAME*, of the mapset ACCOUNT, to be read, with the value of the condition being held by the variable *RESPONSE*.

The condition handling is done by using IF ... statements. If the condition is neither NORMAL nor MAPFAIL the program behaves as if the user had cleared the screen.

If the condition is either NORMAL or MAPFAIL the program saves the value of the key which the user pressed to exit the screen in EXITKEY. In addition, if the condition is NORMAL, the program branches to MODIFYMAP to perform some additional function.

Modifying default CICS exception handling

CICS provides the following EXEC CICS commands which modify the default CICS exception handling and one which modifies the way CICS handles abends:

Note: These commands cannot be used in C, C++, or Java programs. The rest of this chapter is not relevant for these languages.

HANDLE CONDITION

Specify the label to which control is to be passed if a condition occurs.

IGNORE CONDITION

Specify that no action is to be taken if a condition occurs.

HANDLE ABEND

Activate, cancel, or reactivate an exit for abnormal termination processing.

An abend is the commonest way in which CICS handles exception conditions.

The current effect of IGNORE CONDITION, HANDLE ABEND, and HANDLE CONDITION may be suspended by using PUSH HANDLE and reinstated by using POP HANDLE .

You have two ways of passing control to a specified label:

1. Use a HANDLE CONDITION condition(label) command, where condition is the name of an exception condition
2. Use a HANDLE CONDITION ERROR(label) command

The HANDLE CONDITION command sets up some CICS code to name conditions that interest you, and then uses this code to pass control to appropriate sections of your application if those conditions arise. So with an active HANDLE CONDITION command, control goes to whichever label you specified for that particular condition.

The **same** condition can arise, in some cases, on many different commands, and for a variety of reasons. For example, you can get an IOERR condition during file control operations, interval control operations, and others. One of your first tasks, therefore, is to sort out **which command** has raised a particular condition; only when you have discovered that, can you begin to investigate why it has happened. This, for many programmers, is reason enough to start using the RESP option in their new CICS applications. Although you need only one HANDLE CONDITION command to set your error-handling for several conditions, it can sometimes be awkward to pinpoint exactly which of several HANDLE CONDITION commands is currently active when a CICS command fails somewhere in your code.

If a condition which you have not named arises, CICS takes the default action, unless this is to abend the task, in which case it raises the ERROR condition. If you name the condition but leave out its label, any HANDLE CONDITION command for that condition is deactivated, and CICS reverts to taking the default action for it, if and when it occurs.

The need to deal with *all* conditions is a common source of errors when using the HANDLE CONDITION command. When using an unfamiliar command, you should refer to the description of the command in the *CICS Application Programming Reference* manual to find out which exception conditions are possible. Even if you then issue HANDLE commands for all of these, you may not finish all the error-handling code adequately. The outcome is sometimes an error-handling routine that, by issuing a RETURN command, allows incomplete or incorrect data changes to be committed.

The best approach is to use the HANDLE CONDITION command, but to let the system default action take over if you cannot see an obvious way round a particular problem.

Bearing in mind the distinction between an error condition, a condition that merely causes a wait (see page “How CICS keeps track of what to do” on page 268 for examples of conditions that cause a wait), and the special case of the SEND MAP command overflow processing, a HANDLE CONDITION command is active after a HANDLE CONDITION condition(label), or HANDLE CONDITION ERROR(label) command has been run in your application.

If no HANDLE CONDITION command is active for a condition, but one is active for ERROR, control passes to the label for ERROR, if the condition is an error, not a wait.

If you use HANDLE CONDITION commands, or are maintaining an application that uses them, do not include any commands in your error routine that can cause the same condition that gave you the original branch to the routine, because you will cause a loop.

Take special care not to cause a loop on the ERROR condition itself. You can avoid a loop by reverting temporarily to the system default action for the ERROR condition. Do this by coding a HANDLE CONDITION ERROR command with no label specified. At the end of your error processing routine, you can reinstate your error action by including a HANDLE CONDITION ERROR command with the appropriate label. If you know the previous HANDLE CONDITION state, you can do this explicitly. In a general subroutine, which might be called from several different points in your code, the PUSH HANDLE and POP HANDLE command may be useful—see “Using PUSH HANDLE and POP HANDLE commands” on page 271.

Using the HANDLE CONDITION command

Use the HANDLE CONDITION command to specify the label to which control is to be passed if a condition occurs. You must include the name of the condition and you must ensure that the HANDLE CONDITION command is executed before the command that may give rise to the associated condition.

You cannot include more than 16 conditions in the same command. You must specify any additional conditions in further HANDLE CONDITION commands. You can also use the ERROR condition within the same list to specify that all other conditions are to cause control to be passed to the same label.

The HANDLE CONDITION command for a given condition applies only to the program in which it is specified. The HANDLE CONDITION command:

- Remains active while the program is running, or until:
 - An IGNORE CONDITION command for the same condition is met, in which case the HANDLE CONDITION command is overridden
 - Another HANDLE CONDITION command for the same condition is met, in which case the new command overrides the previous one
- Is temporarily deactivated by the NOHANDLE or RESP option on a command

When control passes to another program, by a LINK or XCTL command, the HANDLE CONDITION commands that were active in the calling program are deactivated. When control returns to a program from a program at a lower logical level, the HANDLE CONDITION commands that were active in the higher-level program before control was transferred from it are reactivated, and those in the lower-level program are deactivated. (Refer to “Application program logical levels” on page 566 for information about logical levels.)

The following example shows you how to handle conditions, such as DUPREC, LENGERR, and so on, that can occur when you use a WRITE command to add a record to a data set. Suppose that you want DUPREC to be handled as a special case; that you want standard system action (that is, to terminate the task abnormally) to be taken for LENGERR; and that you want all other conditions to be handled by the error routine ERRHANDL. You would code:

```
EXEC CICS HANDLE CONDITION
        ERROR(ERRHANDL)
        DUPREC(DUPRTN) LENGERR
END-EXEC.
```

In a PL/I application program, a branch to a label in an inactive procedure or in an inactive begin block, caused by a condition, produces unpredictable results.

#

In an assembler language application program, when a branch to a label is caused by a condition, the registers in the application program are restored to their values in the program at the point where the command that caused the condition is issued.

RESP and NOHANDLE options

You can temporarily deactivate the effect of any `HANDLE CONDITION` command by using the `RESP` or `NOHANDLE` option on a command. The way to use these options is described in “Handling exception conditions by in-line code” on page 262. If you do this, you lose the ability to use any system default action for that command. In other words, you have to do your own “catch-all” error processing.

How CICS keeps track of what to do

CICS has a table of the conditions referred to by `HANDLE CONDITION` and `IGNORE CONDITION` commands in your application. Each execution of one of these commands either updates an existing entry in this table, or causes CICS to make a new entry if this is the first time the condition has been quoted in such a command. Each entry tells CICS what to do by indicating one of the three exception-handling states your application can be in, namely:

1. **Let the program continue**, with control coming straight back from CICS to the next instruction following the command that has failed in your program. You can then find out what happened by testing, for example, the `RESP` value that CICS returns after executing a command. The result of this test enables you decide what to do next. For details, see “Handling exception conditions by in-line code” on page 262.

This is the recommended method, which is the approach taken in the “File A” sample programs referred to in the *Sample Applications Guide* and in the COBOL sample application in the *Designing and Programming CICS Applications*. It is also the recommended approach for any new CICS applications. It lends itself to structured code and removes the need for implied `GOTOs` that CICS required in the past.
2. **Pass control to a specified label** if a named condition arises. You do this by using a `HANDLE CONDITION` command or `HANDLE CONDITION ERROR` command to name both the condition and the label of a routine in your code to deal with it. For details, see “Using the `HANDLE CONDITION` command” on page 267 and “Using the `HANDLE CONDITION ERROR` command” on page 269.
3. **Taking the CICS system default action**, where for most conditions, this is to terminate the task abnormally and means that you do nothing by way of testing or handling conditions.

For the conditions `ENQBUSY`, `NOJBUFSP`, `NOSTG`, `QBUSY`, `SESSBUSY`, and `SYSBUSY`, the normal default is to force the task to wait until the required resource (for example, storage) becomes available, and then resume processing the command. You can change this behavior to ignoring the condition by using the `NOSUSPEND` option. For the condition `NOSPACE`, the normal default is to wait if processing a `WRITEQ TS` command, but to abend the task if processing a `WRITEQ TD`, `WRITE`, or `REWRITE` command. Coding the `WRITEQ TS` command with the `NOSUSPEND` option makes it ignore any `NOSPACE` condition that arises. For more information see the *CICS Application Programming Reference* manual.

CICS keeps a table of these conditions for each link level. Essentially, therefore, each program level has its own `HANDLE` state table governing its own condition handling.

This behavior is modified by `HANDLE CONDITION ERROR` and `IGNORE CONDITION`.

Using the HANDLE CONDITION ERROR command

Figure 75 shows the first of only two HANDLE CONDITION commands used in program ACCT01:

```
PROCEDURE DIVISION.  
*  
*   INITIALIZE.  
*   TRAP ANY UNEXPECTED ERRORS.  
*   EXEC CICS HANDLE CONDITION  
*   ERROR(OTHER-ERRORS)  
*   END-EXEC.  
*
```

Figure 75. Trapping the unexpected with the HANDLE CONDITION ERROR command

It passes control to the paragraph at label OTHER-ERRORS if any condition arises for a command that does not specify NOHANDLE or RESP.

The HANDLE CONDITION ERROR command is the first command executed in the procedure division of this COBOL program. This is because a HANDLE CONDITION command must be processed before any CICS command is processed that can raise the condition being handled. Note, however, that your program does not see the effects when it processes the HANDLE CONDITION command; it only sees them later, if and when it issues a CICS command that actually raises one of the named conditions.

In this, and the other ACCT programs, you generally use the RESP option. All the commands specifying the RESP option have been written with a “catch-all” test (IF RESPONSE NOT = DFHRESP(NORMAL) GO TO OTHER-ERRORS) **after** any explicit tests for specific conditions. So any exceptions, other than those you might particularly “expect”, take control to the paragraph at OTHER-ERRORS in each program. Those relatively few commands that do not have RESP on them take control to exactly the same place if they result in any condition other than NORMAL because of this HANDLE CONDITION ERROR command.

Using the IGNORE CONDITION command

Just as you can arrange for control to pass to a particular label for a specific condition with a HANDLE CONDITION command, so you can have the program continue when a specific condition occurs. You do this by setting up an IGNORE CONDITION command to ignore one or more of the conditions that can potentially arise on a command. The IGNORE CONDITION command means that no action is to be taken if a condition occurs; control returns to the instruction following the command and return codes are set in the EIB. The following example ignores the MAPFAIL condition:

```
EXEC CICS IGNORE CONDITION MAPFAIL  
END-EXEC.
```

While a single EXEC CICS command is being processed, it can raise several conditions.³ CICS checks these and passes back to your application program the first one that is not ignored (by your IGNORE CONDITION command). CICS passes back only one exception condition at a time to your application program.

3. For example, you may have a file control command that is not only invalid but also applies to a file not defined in the file control table.

An IGNORE CONDITION command for a given condition applies only to the program you put it in, and it remains active while the program is running, or until a later HANDLE CONDITION command naming the same condition is met, in which case the IGNORE CONDITION command is overridden.

You can choose an IGNORE CONDITION command if you have a program reading records that are sometimes longer than the space you provided, but you do not consider this an error and do not want anything done about it. You might, therefore, code IGNORE CONDITION LENGERR before issuing READ commands.

You can also use an IGNORE CONDITION ERROR command to catch any condition considered as an error for which there is no currently active HANDLE CONDITION command that includes a label. When an error occurs, control is passed to the next statement and it is up to the program to check for return codes in the EIB. See page “How CICS keeps track of what to do” on page 268 for examples of conditions that are **not** considered as errors.

You can also switch from ignoring a condition to handling it, or to using the system default action. For example, you could code:

```
*      MIXED ERROR PROCESSING
      EXEC CICS IGNORE CONDITION LENGERR
      END-EXEC.
:
      EXEC CICS HANDLE CONDITION DUPREC(DUPRTN)
      LENGERR
      ERROR(ERRHANDL)
      END-EXEC.
```

Because this code initially ignores condition LENGERR, nothing happens if the program raises a LENGERR condition; the application simply continues its processing. Of course, if the fact that LENGERR has arisen means that the application cannot sensibly continue, you have a problem.

Later in the code, you can explicitly set condition LENGERR to the system default action by naming it in a HANDLE CONDITION command without a label. When this command has been executed, the program no longer ignores condition LENGERR, and if it subsequently occurs, it now causes the system default action. The point about mixing methods is that you can, and that each condition is treated separately.

You cannot code more than 16 conditions in the same command. You must specify additional conditions in further IGNORE CONDITION commands.

Using the HANDLE ABEND command

Note to Java, C and C++ programmers

Handle ABEND is not applicable to Java programs. Although HANDLE ABEND is supported in C and C++ when used with the PROGRAM option, it is not helpful in the context of this chapter because exception conditions in C and C++ programs do not cause abends.

The HANDLE ABEND command activates or reactivates a program-level abend exit within your application program; you can also use this command to cancel a previously activated exit. For more information see the *CICS Application Programming Reference* manual .

CICS does not allow the use of HANDLE ABEND LABEL in Assembler programs that do not use DFHEIENT and DFHEIRET. Assembler programs that use the Language Environment stub CEESTART should either use HANDLE ABEND PROGRAM or a Language Environment service such as CEEHDLR.

HANDLE ABEND lets you supply your own code to be executed when an abend is processed. This means that your application can cope with the abnormal situation in an orderly manner and carry on executing. You provide the user exit programs and rely on CICS calling them when required.

The flow of control during abend processing is shown in Figure 76 on page 276.

Using PUSH HANDLE and POP HANDLE commands

PUSH HANDLE

Suspends the current effect of HANDLE CONDITION, IGNORE CONDITION, HANDLE ABEND and HANDLE AID commands.

POP HANDLE

Reinstates the effect of HANDLE CONDITION, IGNORE CONDITION, HANDLE ABEND and HANDLE AID commands to what they were before the previous PUSH HANDLE was called.

CICS also keeps a table of conditions for each PUSH HANDLE command which has not been countermanded by a matching POP HANDLE command.

When each condition occurs, CICS performs the following sequence of tests:

1. If the command has the RESP or NOHANDLE option, control returns to the next instruction in your application program. Otherwise, CICS scans the condition table to see what to do.
2. If an entry for the condition exists, this determines the action.
3. If no entry exists and the default action for this condition is to suspend execution:
 - a. If the command has the NOSUSPEND or NOQUEUE option, control returns to the next instruction.
 - b. If the command does not have one of these options, the task is suspended.
4. If no entry exists and the default action for this condition is to abend, a second search is made, this time for the ERROR condition:
 - a. If found, this entry determines the action.
 - b. If ERROR cannot be found, the task is abended. You can choose to handle abends. For information about the HANDLE ABEND command, see the *CICS Application Programming Reference*.

Note: The OVERFLOW condition on aSEND MAP command is an exception to the above rules. See the *CICS Application Programming Reference* for more information.

The commands ALLOCATE, ENQ, GETMAIN, WRITE JOURNALNAME, WRITE JOURNALNUM, READQ TD, and WRITEQ TS can all raise conditions for which the default action is to suspend your application program until the specified resource becomes available. So, on these commands, you have the NOSUSPEND option to inhibit this waiting and return immediately to the next instruction in your application program.

Some conditions can occur during the execution of a number of unrelated commands. If you want the same action for all occurrences, code a single `HANDLE CONDITION` command at the start of your program.

Note: As using `RESP` implies `NOHANDLE`, be careful when using `RESP` with the `RECEIVE` command, because it overrides the `HANDLE AID` command as well as the `HANDLE CONDITION` command. This means that PF key responses are ignored, and is the reason for testing them earlier in the `ACCT` code. See “Using the `HANDLE AID` command” on page 499.

Chapter 18. Abnormal termination recovery

CICS provides a program-level abend exit facility so that you can write exits of your own which can receive control during abnormal termination of a task. The “cleanup” of a program that has started but not completed normally is an example of a function performed by such an abend exit.

Here are some causes of abnormal terminations:

- A user request by, for example:
`EXEC CICS ABEND ABCODE(...)`
- A CICS request as a result of an invalid user request. For example, an invalid FREEMAIN request gives the transaction abend code ASCF.
- A program check, in which case the system recovery program (DFHSRP) is driven, and the task abends with code ASRA.
- An operating system abend, in which case DFHSRP is driven, and the task abends with code ASRB.
- A looping task, in which case DFHSRP is driven, and the task abends with code AICA.

Note: If an ASRB or ASRA is detected in CICS code, CICS produces a dump before calling your HANDLE ABEND exit.

See the *CICS Problem Determination Guide* for full details about fixing problems, and see the *CICS Messages and Codes* for information about the transaction abend codes for abnormal terminations that are initiated by CICS, their meanings, and your responses.

The HANDLE ABEND command activates or reactivates a program-level abend exit within your application program; you can also use this command to cancel a previously activated exit.

When activating an exit, you must use the PROGRAM option to specify the name of a program to receive control, or (except for C, C++, and PL/I programs) the LABEL option to specify a routine label to which control branches when an abnormal termination condition occurs. Using an ON ERROR block in PL/I is the equivalent of using the HANDLE ABEND LABEL command.

A HANDLE ABEND command overrides any preceding such command in any application program at the same logical level. Each application program of a transaction can have its own abend exit, but only one abend exit at each logical level can be active. (Logical levels are explained in Chapter 45, “Program control,” on page 565.)

When a task terminates abnormally, CICS searches for an active abend exit, starting at the logical level of the application program in which the abend occurred, and proceeding to successively higher levels. The first active abend exit found, if any, is given control. This procedure is shown in Figure 76 on page 276, which also shows how subsequent abend processing is determined by the user-written abend exit.

If CICS finds no abend exit, it passes control to the abnormal condition program to terminate the task abnormally. This program invokes the user replaceable program

error program, DFHPEP. See the *CICS Customization Guide* for programming information about how to customize DFHPEP.

CICS deactivates the exit upon entry to the exit routine or program to prevent recursive abends in an abend exit. If you wish to retry the operation, you can branch to a point in the program that was in control at the time of the abend and issue a HANDLE ABEND RESET command to reactivate the abend exit. You can also use this command to reactivate an abend exit (at the logical level of the issuing program) that was canceled previously by a HANDLE ABEND CANCEL command. You can suspend the HANDLE ABEND command by means of the PUSH HANDLE and POP HANDLE commands as described in “Using PUSH HANDLE and POP HANDLE commands” on page 271.

Note that when an abend is handled, the dynamic transaction backout program is not be invoked. If you need the dynamic transaction backout program, you take an implicit or explicit syncpoint or issue SYNCPOINT ROLLBACK or issue an ABEND command.

Where the abend is the result of a failure in a transaction running in an IRC-connected system, for example AZI2, the syncpoint processing may abend ASP1 if it attempts to use the same IRC connection during its backout processing.

The HANDLE ABEND command cannot intercept ASPx or APSJ abend codes.

This chapter describes:

- “Creating a program-level abend exit”
- “Retrying operations” on page 275
- “Trace” on page 276
- “Monitoring” on page 277
- “Dump” on page 278

Creating a program-level abend exit

You can either define abend exits by using RDO or by using the program autoinstall exit. If you use the autoinstall method, the program definition is not available at the time of the HANDLE ABEND. This may mean that a program functions differently the first time it is invoked. If the program is not defined at the time the HANDLE ABEND is issued, and program autoinstall is active, the security check on the name of the program is the only one which takes place. Other checks occur at the time the abend program is invoked. If the autoinstall fails, the task abends APCT and control is passed to the next higher level.

Abend exit programs can be coded in any supported language, but abend exit routines must be coded in the same language as their program.

For abend exit routines, the addressing mode and execution key are set to the addressing mode and execution key in which the HANDLE ABEND command has been issued.

Upon entry to an abend exit program, no addressability can be assumed other than that normally assumed for any application program coded in that language. There are no register values for C, C++, or PL/I languages as these languages do not support HANDLE ABEND label.

Upon entry to an abend exit routine, the register values are:

COBOL

Control returns to the HANDLE ABEND command with the registers restored; a COBOL GOTO is then executed.

Assembler

Reg 15

Abend label.

Reg 0-14

Contents at the time of the last CICS service request.

There are three means of terminating processing in an abend exit routine or program, as listed below. It is recommended that when abend routines and programs are called by CICS internal logic they should terminate with an abend because further processing is likely to cause more problems.

1. Using a RETURN command to indicate that the task is to continue running with control passed to the program on the next higher logical level. If no such program exists, the task is terminated normally, and any recoverable resources are committed.
2. Using an ABEND command to indicate that the task is to be abnormally terminated with control passed either to an abend exit specified for a program on a higher logical level or, if there is not one, to the abnormal condition program for abnormal termination processing.
3. Branching to retry an operation. When you are using this method of retrying an operation, and you want to reenter the original abend exit routine or program if a second failure occurs, the abend exit routine or program should issue the HANDLE ABEND RESET command before branching. This is because CICS has disabled the exit routine or program to prevent it reentering the abend exit.

In the case of an abend caused by a timeout on an outstanding RECEIVE command, it is important to let the CICS abend continue, so that CICS can cancel the RECEIVE.

Retrying operations

If an abend occurs during the invocation of a CICS service, you should be aware that issuing a further request for **the same service** may cause unpredictable results, because the reinitialization of pointers and work areas, and the freeing of storage areas in the exit routine, may not have been completed.

You should not try to recover from ATNI or ATND abends by attempting further I/O operations. Either of these abends results in a TERMERR condition, requiring the session to be terminated in all cases. You should not try to issue terminal control commands while recovering from an AZCT abend, or an AZIG abend, as CICS has not fully cleaned up from the RTIMOUT, and an indefinite wait can occur.

If intersystem communication is being used, an abend in the remote system might cause a branch to the specified program or label, but subsequent requests to use the same resource in the remote system might fail. If an abend occurs as a result of a failure in the connection to the remote system, subsequent requests to use *any* resources in the remote system might fail.

If an abend occurs as a result of a BMS command, control blocks are not tidied up before control is returned to the BMS program, and results are unpredictable if the command is retried.

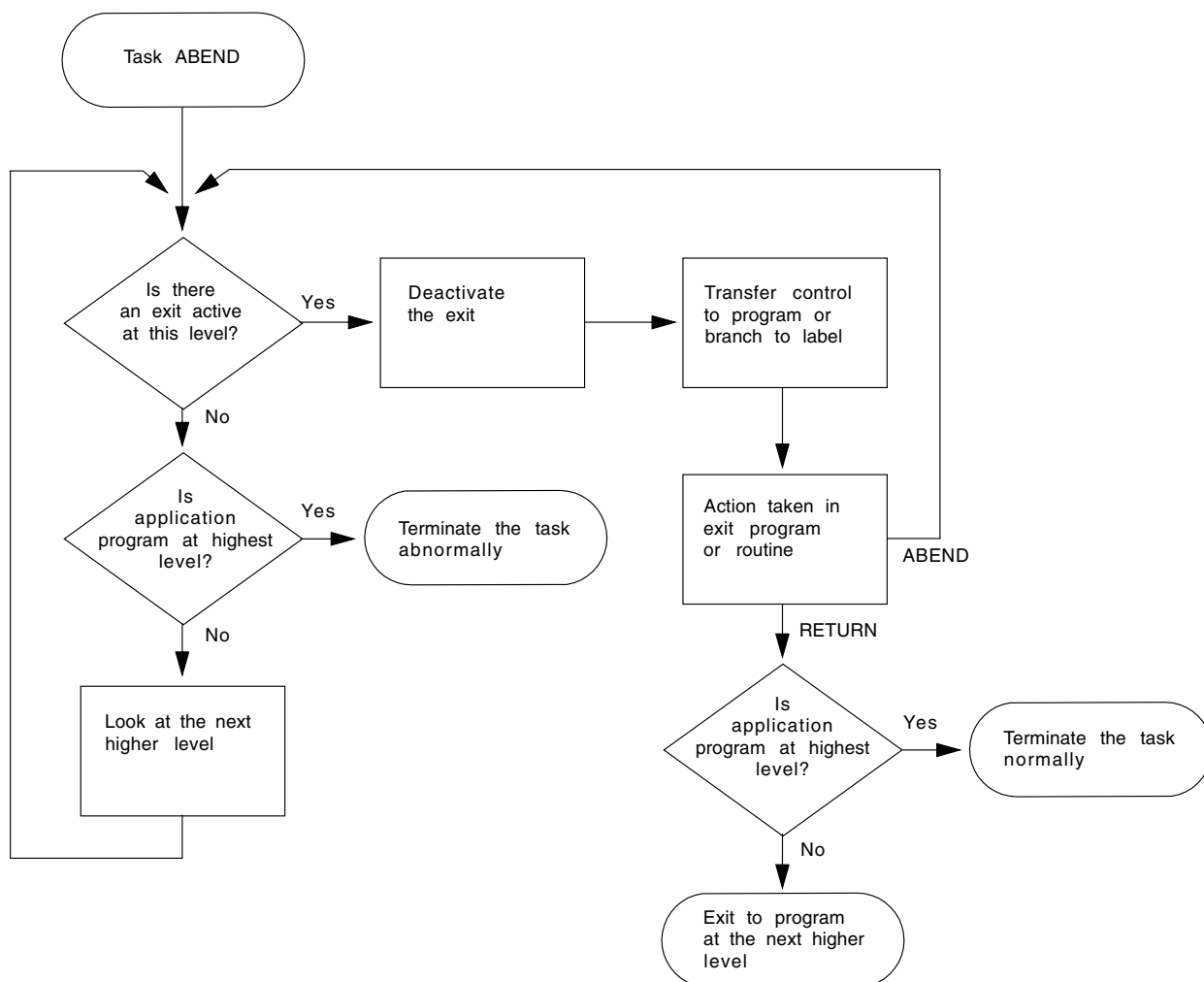


Figure 76. ABEND exit processing

Trace

CICS trace is a debugging aid for application programmers, system programmers, and IBM field engineers. It produces trace entries in response to trace commands. The trace entries can be sent to any trace destination that is currently active. The destinations are:

- Internal trace table
- Auxiliary trace data set
- Generalized trace facility (GTF) data set

For information about trace destinations, see the *CICS Problem Determination Guide*.

You can:

- Specify user trace entry points (ENTER TRACENUM). (The earlier ENTER TRACEID command is supported for compatibility purposes. See the *CICS for MVS/ESA 4.1 Migration Guide* for details.)
- Switch CICS internal trace on or off using the SET TRACEDEST, SET TRACEFLAG, and SET TRACETYPE commands.

Trace entry points

The points at which trace entries are produced during CICS operation are of four types: system trace entry points, user trace entry points, exception trace entry points, and user exception trace entry points. See the *CICS Problem Determination Guide* for more information about tracing.

System trace entry points

These are points within CICS at which trace control requests are made. The most important system trace entry points for application programmers are for the EXEC interface program. These produce entries in the trace table whenever a CICS command is processed.

Two trace entries are made: the first when the command is issued, and the second when CICS has performed the required function and is about to return control to your application program. Between them, these two trace entries allow you to trace the flow of control through an application, and to check which exception conditions, if any, occurred during its execution. The ABEND, RETURN, TRACEFLAG, and XCTL commands produce single entries only.

User trace entry points

These are additional points within your application program that you can include in the trace table to allow complete program debugging. For example, you could specify an entry for a program loop containing a counter value showing the number of times that the loop had been entered.

A trace entry is produced wherever the ENTER TRACENUM command is run. Each trace entry request, which can be given a unique identifier, causes data to be placed in the trace table.

Exception trace entry points

These are additional points where CICS has detected an exception condition. These are made from specific points in the CICS code, and data is taken from areas that might provide some information about the cause. Exception trace entry points do not have an associated “level” attribute; trace calls are only ever made from them when exception conditions occur.

User exception trace entry points

These are trace entries that are always written to the internal trace table (even if internal tracing is set off), but are written to other destinations only if they are active. You can identify them by the character string *EXCU in any formatted trace output produced by the CICS utility programs. See the *CICS Problem Determination Guide* for general information about user exception trace entry points; programming information is in the *CICS Customization Guide*.

Monitoring

CICS monitoring provides information about the performance of your application transactions.

You should use the MONITOR command for user event monitoring points.

In addition to the monitoring data collected from a system defined elsewhere, monitoring points (EMPs) within CICS, a user application program can contribute data to user fields within the CICS monitoring records. You can do this by using the MONITOR POINT command to invoke user-defined EMPs. At each of these EMPs,

you can add or change up to 4096 bytes of your own data in each performance monitoring record. In those 4096 bytes, you can have any combination of the following:

- In the range 0 through 256 counters
- In the range 0 through 256 clocks
- A single 8192-byte character string

For example, you could use these user EMPs to count the number of times a certain event occurs, or to time the interval between two events. For programming information about monitoring, see the *CICS Customization Guide*; for background information, see the *CICS Performance Guide*.

Dump

CICS dump allows you to specify areas of main storage to be dumped, by means of the DUMP TRANSACTION , onto a sequential data set, which can be either on disk or tape.

The PERFORM DUMP command allows you to request a system dump. See the *CICS System Programming Reference* manual for programming information about PERFORM DUMP.

You can format the contents of the dump data set and you can print them offline using the CICS dump utility program (DFHDU640) for transaction dumps or the interactive problem control system (IPCS) for system dumps. Instructions on using these programs are given in the *CICS Operations and Utilities Guide*.

Only one dump control command is processed at a time. If you issue additional dump control commands, while another task is taking a transaction dump, activity within the tasks associated with those commands is suspended until the dump is completed. Remaining dump commands are processed in the order in which they are made. Using the DUMP TRANSACTION command causes some fields (for example, EIBFN and EIBRCODE) in the EIB and the TCA to be overwritten. See the *CICS Application Programming Reference* manual for programming information about DUMP TRANSACTION.

Options on the DUMP TRANSACTION command allow you to dump the following areas of main storage in various combinations:

- Task-related storage areas: selected main storage areas related to the requesting task. You would normally use a dump of these areas to test and debug your application program. (CICS automatically provides this service if the related task is terminated abnormally.)
- CICS control tables:
 - File control table (FCT)
 - Program control table (PCT)
 - Processing program table (PPT)
 - System initialization table (SIT)
 - Terminal control table (TCT)

A dump of these tables is typically the first dump taken in a test in which the base of the test must be established; subsequent dumps are usually of the task-related storage type.

- It is sometimes appropriate during execution of a task to have a dump of both task-related storage areas and CICS control tables. Specifying one CICS control tables dump and a number of task-related storage dumps is generally more

efficient than specifying a comparable number of complete dumps. However, you should not use this facility excessively because CICS control tables are primarily static areas.

- In addition, the DUMP TRANSACTION command used with the three options, SEGMENTLIST, LENGTHLIST, and NUMSEGMENTS, allows you to dump a series of task-related storage areas simultaneously.

Program storage is not dumped for programs defined with the attribute RELOAD(YES).

You also get a list of the CICS nucleus modules and active PPT programs, indexed by address, at the end of the printed dump.

Chapter 19. The QUERY SECURITY command

QUERY SECURITY is effective with RACF® or any equivalent external security manager (ESM). You can use this command to query whether the terminal user has access to resources that are defined to the external security manager. These can be:

- Resources in CICS resource classes
- Resources in user-defined resource classes

The terminal user in this context is the user invoking the transaction that contains the QUERY SECURITY command.

Java and C++

The application programming interface described in this chapter is the EXEC CICS API, which is not used in Java programs. For information about Java programs using the JCICS classes to access security control services, see the *Java Applications in CICS* component of the CICS Information Center and the JCICS Javadoc html documentation. For information about C++ programs using the CICS C++ classes, see the *CICS C++ OO Class Libraries* manual.

In response to a QUERY SECURITY command, CICS returns information about the terminal user's security authorizations. CICS obtains this information from the external security manager. You can code the application to proceed in different ways depending on the user's permitted accesses.

You specify the type of resource that you are querying by the CICS resource type name. For example, if you want to query a user's authorization to access a file, you can specify RESTYPE('FILE'). To identify a particular file within the type, you specify the RESID parameter.

Using the QUERY SECURITY command

A typical use of the QUERY SECURITY command is to check whether a user is authorized to use a particular transaction **before** displaying the transaction code in a menu.

Security protection at the record or field level

Another use for QUERY SECURITY is to enable you to control access to data at the record or field level. The normal CICS resource security checking for file resources, for example, works only at the file level. To control access to individual records, or even fields within records, you can use QUERY SECURITY. For this purpose, your security administrator must define resource profile names, with appropriate access authorizations, for the records or fields that you want to protect. These profiles are defined in user resource classes defined by the administrator, **not** in CICS resource classes.

To query these classes and resources, the QUERY SECURITY command uses the RESCLASS and RESID options (RESCLASS and RESTYPE are mutually exclusive options). You can use the CVDA values returned by QUERY SECURITY to determine whether to access the record or field.

CICS-defined resource identifiers

In all cases except for the SPCOMMAND resource type, the resource identifiers are user-defined. However, for the SPCOMMAND type, the identifiers are fixed by CICS. The *CICS RACF Security Guide* details the possible RESID values for the SPCOMMAND resource type.

SEC system initialization parameter

The setting of the SEC system initialization parameter affects the CVDA values returned by the QUERY SECURITY command. The SEC system initialization parameters are described in more detail in the *CICS RACF Security Guide*.

Programming hints

- A transaction can use the QUERY SECURITY command to query a number of resources in order to prepare a list of resources to which the terminal user has access. The use of this technique could generate up to four resource violation messages for each query on a resource that the transaction is not authorized to access. These messages appear on the system console, the CSCS TD queue, and the SMF log data set. If you want to suppress these messages, code NOLOG in the QUERY SECURITY command.
- If a transaction accesses the same resource many times in one execution, you can probably improve performance by defining the transaction with RESSEC(NO) in the transaction resource definition. You can then code the transaction to issue a single QUERY SECURITY command, and to permit access to the resource according to the CVDA values returned. For detailed guidance, see the *CICS RACF Security Guide*.

Part 4. Data Management

Chapter 20. An overview of file control.	287
VSAM data sets	287
Key-sequenced data set (KSDS)	287
Entry-sequenced data set (ESDS)	288
Relative record data set (RRDS)	288
Empty data sets	289
VSAM alternate indexes	289
Accessing files in RLS mode	289
Some RLS limitations	290
BDAM data sets	290
CICS shared data tables	291
Coupling facility data tables	292
Coupling facility data table models	293
Techniques for sharing data	294
Using CICS commands to read records	296
Direct reading (using READ command)	297
Direct reading from a KSDS	297
Direct reading from an ESDS	297
Direct reading from an RRDS	297
Direct reading by way of a path	297
Read integrity (in RLS mode)	298
Sequential reading (browsing)	298
Browsing through a KSDS	299
Browsing through an ESDS	300
Browsing through an RRDS	300
Browsing using a path	300
Browse integrity (in RLS mode)	300
Ending the browse	300
Simultaneous browse operations	300
Skip-sequential processing	300
Using CICS commands to update records	301
Using CICS commands to delete records	302
Deleting single records	302
Updating and deleting records in a browse (VSAM RLS only)	302
Deleting groups of records (generic delete)	303
Read integrity	303
Using CICS commands to add records	303
Adding to a KSDS	303
Adding to an ESDS	304
Adding to an RRDS	304
Records that are already locked	304
Specifying record length	304
Sequential adding of records (WRITE MASSINSERT command)	304
File control command options	305
The RIDFLD option	305
The INTO and SET options	305
The FROM option	306
The TOKEN option	306
KEYLENGTH option for remote data sets	306
Transaction deadlocks	307
VSAM-detected deadlocks (RLS only)	308
Rules for avoiding deadlocks	309

Chapter 21. File control—VSAM considerations	311
VSAM record identification	311
Key.	311
Relative byte address (RBA) and relative record number (RRN)	311
RBA	311
RRN	312
Locking of VSAM records in recoverable files	312
Update locks and delete locks (non-RLS mode only)	312
RLS Record level locking	313
Exclusive locks and shared locks.	313
Exclusive locks	314
Shared locks	314
Lock duration	314
Active and retained states for locks	315
Conditional VSAM file update requests	316
File control implementation of NOSUSPEND	316
CICS locking for writing to ESDS.	316
Chapter 22. File control—BDAM considerations	319
Identifying BDAM records	319
Block reference subfield	319
Physical key subfield	319
Deblocking argument subfield	319
Browsing records from BDAM data sets	320
Adding records to BDAM data sets	321
Updating records from BDAM data sets	322
Chapter 23. Database control	323
DL/I databases	323
DB2 databases	323
Requests to DB2.	323
Chapter 24. CICS documents	325
Using the DOCUMENT programming interface.	325
Creating a document	325
The BINARY option.	325
The TEXT option.	325
The FROMDOC option	326
Setting up document templates	326
Templates in a partitioned data set	326
Templates in a CICS program	327
Templates in CICS files, z/OS UNIX System Services HFS files, temporary storage, or transient data	327
Creating templates in exit programs.	328
Programming with documents and document templates	329
Symbols and symbol lists	329
Setting symbol values	330
Embedded template commands	332
Using templates in your application	333
The lifespan of a document.	334
Retrieving the document without control information	335
Constructing a document.	336
Using Bookmarks	338
Replacing data in the document	339
Code page conversion for documents	339

Chapter 25. Named counter servers	341
The named counter fields	341
Named counter pools	342
Named counter options table	342
Using the named counter EXEC interface	343
Using the named counter CALL interface	345
Application programming considerations	345
Syntax	347
Checking for result overflow.	353
Example of DFHNCTR calls with null parameters.	353
Return codes	355
Named counter recovery	358

Chapter 20. An overview of file control

CICS data management services have traditionally been known as **CICS file control**. CICS file control I offers you access to data sets that are managed by either the virtual storage access method (VSAM) or the basic direct access method (BDAM).

CICS file control lets you read, update, add, and browse data in VSAM and BDAM data sets and delete data from VSAM data sets. You can also access CICS shared data tables and coupling facility data tables using file control.

A CICS application program reads and writes its data in the form of individual records. Each read or write request is made by a CICS command.

To access a record, the application program must identify both the record and the data set that holds it. It must also specify the storage area into which the record is to be read or from which it is to be written.

This chapter describes:

- “VSAM data sets”
- “BDAM data sets” on page 290
- “CICS shared data tables” on page 291
- “Coupling facility data tables” on page 292
- “Techniques for sharing data” on page 294
- “Using CICS commands to read records” on page 296
- “Using CICS commands to update records” on page 301
- “Using CICS commands to delete records” on page 302
- “Using CICS commands to add records” on page 303
- “File control command options” on page 305
- “Transaction deadlocks” on page 307

VSAM data sets

CICS supports access to the following types of data set:

- Key-sequenced data set (KSDS)
- Entry-sequenced data set (ESDS)
- Relative record data set (RRDS) (both fixed and variable record lengths)

VSAM data sets are held on direct access storage devices (DASD) auxiliary storage. VSAM divides its data set storage into control areas (CA), which are further divided into control intervals (CI). Control intervals are the unit of data transmission between virtual and auxiliary storage. Each one is of fixed size and, in general, contains a number of records. A KSDS or ESDS can have records that extend over more than one control interval. These are called spanned records.

Key-sequenced data set (KSDS)

A **key-sequenced data set** has each of its records identified by a key. (The **key** of each record is simply a field in a predefined position within the record.) Each key must be unique in the data set.

When the data set is initially loaded with data, or when new records are added, the logical order of the records depends on the collating sequence of the key field. This also fixes the order in which you retrieve records when you browse through the data set.

To find the physical location of a record in a KSDS, VSAM creates and maintains an **index**. This relates the key of each record to the record's relative location in the data set. When you add or delete records, this index is updated accordingly.

With releases of DFSMS/MVS 1.4 and later, a data set can be greater than 4GB in size if it is defined as extended format and extended addressability in the storage class. CICS supports, in both RL and non-RLS mode, KSDS data sets that are defined with these extended attributes.

Entry-sequenced data set (ESDS)

An **entry-sequenced data set** is one in which each record is identified by its relative byte address (RBA).

Records are held in an ESDS in the order in which they were first loaded into the data set. New records added to an ESDS always go after the last record in the data set. You may not delete records or alter their lengths. After a record has been stored in an ESDS, its RBA remains constant. When browsing, records are retrieved in the order in which they were added to the data set.

With releases of DFSMS/MVS 1.5 and later, a data set can be greater than 4 GB in size if it is defined as extended format and extended addressability in the storage class. However, CICS does not support ESDS data sets that are defined with these extended attributes. Attempts to open data sets defined with the extended attribute fail with error message DFHFC0966, codes 8504, 0008, and 0068 for non-RLS access and message DFHFC0905 for RLS access.

#

Relative record data set (RRDS)

A **relative record data set** has records that are identified by their relative record number (RRN). The first record in the data set is RRN 1, the second is RRN 2, and so on.

Records in an RRDS can be fixed or variable length records, and the way in which VSAM handles the data depends on whether the data set is a fixed or variable RRDS. A fixed RRDS has fixed-length slots predefined to VSAM, into which records are stored. The length of a record on a fixed RRDS is always equal to the size of the slot. VSAM locates records in a fixed RRDS by multiplying the slot size by the RRN (which you supply on the file control request), to calculate the byte offset from the start of the data set.

A variable RRDS, on the other hand, can accept records of any length up to the maximum for the data set. In a variable RRDS VSAM locates the records by means of an index.

A fixed RRDS generally offers better performance. A variable RRDS offers greater function.

With releases of DFSMS/MVS 1.5 and later, a data set can be greater than 4 GB in size if it is defined as extended format and extended addressability in the storage class. CICS supports access to extended RRDS or VRRDS datasets if you use an RRN that can be specified in a four-byte RRN field to access the records that reside beyond the 4 GB boundary.

#

Empty data sets

An empty data set is a data set that has not yet had any records written to it. VSAM imposes several restrictions on an empty data set that is opened in non-RLS access mode. However, CICS hides all these restrictions from you, allowing you to use an empty data set in the same way as a data set that contains data, regardless of the access mode.

VSAM alternate indexes

Sometimes you want to access the same set of records in different ways. For example, you may have records in a personnel data set that have as their key an employee number. No matter how many Smiths you have, each of them has a unique employee number. Think of this as the primary key.

If you were producing a telephone directory from the data set, you would want to list people by name rather than by employee number. You can identify records in a data set with a secondary (alternate) key instead of the primary key described above. So the primary key is the employee number, and the employee name is the **alternate key**. Alternate keys are just like the primary key in a KSDS—fields of fixed length and fixed position within the record. You can have any number of alternate keys per base file and, unlike the primary or base key, alternate keys need not be unique.

To continue the personnel example, the employee's department code might be defined as a further alternate key.

VSAM allows KSDS and ESDS (but not RRDS) data sets to have alternate keys. When the data set is created, one secondary or **alternate index** is built for each alternate key in the record and is related to the primary or base key. To access records using an alternate key, you must define a further VSAM object, an **alternate index path**. The path then behaves as if it were a KSDS in which records are accessed using the alternate key.

When you update a record by way of a path, the corresponding alternate index is updated to reflect the change. However, if you update the record directly by way of the base, or by a different path, the alternate index is only updated if it has been defined to VSAM (when created) to belong to the **upgrade set** of the base data set. For most applications, you probably want your alternate index to be in the upgrade set.

A CICS application program disregards whether the file it is accessing is a path or the base. In a running CICS system, access to a single base data set can be made by way of the base and by any of the paths defined to it, if each access route is defined in the file control table (FCT).

It is also possible for a CICS application program to access a file that has been directly defined as an alternate index rather than a path. This results in index data being returned to the application program rather than file data. This operation is not supported for files opened in record-level sharing (RLS) mode.

Accessing files in RLS mode

Record-level sharing (RLS) is a VSAM function, provided by DFSMS Version 1 Release 3 and later releases, that enables VSAM data to be shared, with full update capability, between many applications running in many CICS regions.

CICS can define a further structure on top of BDAM data sets, introducing the concept of blocked-data sets:



Figure 78. Blocked-data set

The data portion of the physical record is viewed as a block containing logical records. CICS supports the retrieval of logical records from the data part of the physical record. CICS also supports unblocked records (where the structure reverts to the original BDAM concept of one logical record per physical record).

To retrieve data from a physical record in a BDAM file under CICS, a record identification field (RIDFLD) has to be defined to specify how the physical record should be retrieved. This may be done using the physical key, by relative address, or by absolute address.

If the data set is defined to CICS as being blocked, individual records within the block can be retrieved (deblocked) in two addressing modes: by key or by relative record.

Deblocking by key uses the key of the logical record (that is, the key contained in the logical record) to identify which record is required from the block. Deblocking by relative record uses the record number in the block, relative to zero, of the record to be retrieved.

You specify the key or relative record number used for deblocking in a subfield of the RIDFLD option used when accessing CICS BDAM files. The addressing mode for CICS BDAM files is set in the FCT using the RELTYPE keyword.

For more details about record identification and BDAM record access, see Chapter 22, “File control—BDAM considerations,” on page 319.

CICS shared data tables

The file control commands can access shared data tables. Shared data tables offer a method of constructing, maintaining, and gaining rapid access to data records contained in tables held in virtual storage, above the 16MB line. Each shared data table is associated with a VSAM KSDS, known as its **source data set**. For more information about shared data tables, see the *CICS Shared Data Tables Guide*.

A table is defined using the CEDA DEFINE FILE panel. When a table is opened, CICS builds it by extracting data from the table’s corresponding source data set and loading it into virtual storage above the 16MB line.

CICS supports two types of shared data table, as follows:

CICS-maintained tables (CMTs)

This type of data table is kept in synchronization with its source data set by CICS. All changes to the data table are reflected in the source data set. Similarly all changes to the source data set are reflected in the data table.

Note that the source for a CICS-maintained data table cannot be a file opened in RLS access mode.

User-maintained tables (UMTs)

This type of data table is completely detached from its source data set after it has been loaded. Changes to the table are **not** automatically reflected in the source data set.

The full file control API appropriate to VSAM KSDS data sets is supported for CICS-maintained data tables. Requests that cannot be satisfied by reference to the data table result in calls to VSAM to access the source data set. Tables defined to be recoverable are supported with full integrity.

A subset of the file control API is supported for user-maintained tables. For programming information about the commands, you should see the *CICS Application Programming Reference* manual where they are listed separately under the file control command name, followed by UMT. For example, the information on writing a record to a user-maintained table is given under WRITE(UMT). A table defined as recoverable participates in dynamic transaction backout but is not recovered at restart or XRF takeover.

Coupling facility data tables

The CICS file control commands can access coupling facility data tables (CFDTs). Coupling facility data tables provide a method of file data sharing, without the need for a file-owning region, and without the need for VSAM RLS support. CICS coupling facility data table support is designed to provide rapid sharing of working data across a sysplex, with update integrity. The data is held in a coupling facility, in a table that is similar in many ways to a shared user-maintained data table. A coupling facility data table is different from a UMT in one important respect in that initial loading from a VSAM source data set is optional. You can specify LOAD(NO) and load the table by writing data directly from a user application program. The API used to store and retrieve the data is based on the file control API used for user-maintained data tables. Read access and write access to CFDTs have similar performance, making this form of table particularly useful for informal shared data. Informal shared data is characterised as:

- Data that is relatively short-term in nature (it is either created as the application is running, or is initially loaded from an external source)
- Data volumes that are not usually very large
- Data that needs to be accessed fast
- Data of which the occasional loss can be tolerated by user applications
- Data that commonly requires update integrity.

Typical uses might include sharing scratchpad data between CICS regions across a sysplex, or sharing of files for which changes do not have to be permanently saved. There are many different ways in which applications use informal shared data, and most of these could be implemented using coupling facility data tables. Coupling facility data tables are particularly useful for grouping data into different tables, where the items can be identified and retrieved by their keys. For example, you could use a record in a coupling facility data table to maintain the next free order number for use by an order processing application. Other examples are:

- Look-up tables of telephone numbers or the numbers of stolen credit cards
- Working data consisting of a few items, such as a subset of customers from a customer list

- Information that is specific to the user of the application, or that relates to the terminal from which the application is being run
- Data extracted from a larger file or database for further processing.

Coupling facility data tables allow various types of access to your informal data: read-only, single updater, multiple updaters, sequential access, random access, random insertion and deletion.

For many purposes, because it is global in scope, coupling facility data tables can offer significant advantages over resources such as the CICS common work area (CWA).

To an application program, a CFDT appears much like a sysplex-wide user-maintained data table: a CFDT is accessed using the same subset of the API as a UMT (that is, the full file control API except for the MASSINSERT and RBA options). However, a CFDT is restricted to a maximum key-length of 16 bytes.

Note the following comparisons with user-maintained data tables:

- Updates to a CFDT, like updates to a UMT, are not reflected in the base VSAM data set (if the table was initially loaded from one). Updates are made to the CFDT only.
- A CFDT is loaded once only, when the table is first created in the coupling facility data table, and remains in existence in the coupling facility, even when the last file referring to the CFDT is closed (whereas a UMT is deleted each time the owning region terminates). You can force a reload of a CFDT from the original source data set only by first deleting the table from the CFDT pool, using a CFDT server DELETE TABLE command. The first file opened against the CFDT after the delete operation causes the server to reload the table.

Note: A coupling facility data table pool is defined as a coupling facility list structure, and can hold more than one data table (see the *CICS System Definition Guide* for information about creating a list structure for coupling facility data tables).

- The access rules for a UMT that is in the course of loading allow any direct read request to be satisfied either from the table (if the record has already been loaded) or from the source data set, but reject any update request, or imprecise read or browse request, with the LOADING condition. For a CFDT, any request is allowed during loading, but requests succeed only for records that are within the key range already loaded.

Coupling facility data table models

There are two models of coupling facility data table:

Contention model

This gives optimal performance, but requires programs that are written to handle the situation where the data has been changed since it issued a read-for-update request. The new CHANGED response can occur on a REWRITE or DELETE command. There is also a new use for the existing NOTFND response, which may be returned to indicate to the application program that the record has been deleted since the program issued the read-for-update request.

Note: It might be possible to use existing programs with the contention model if you are sure they cannot receive the CHANGED or NOTFND exceptions on a REWRITE or DELETE. An example of this

could be where an application program operates only on records that relate to the user of the program, and therefore no other user could be updating the same records.

Locking model

This model is API-compatible with existing programs that conform to the UMT subset of the file control API. The locking model can be:

Non-recoverable

For updates to non-recoverable CFDTs, locks do not last until syncpoint (they are released on completion of the file control request) and updates are not backed out if a unit of work fails

Recoverable

CFDTs are recoverable in the event of a unit of work failure, and in the event of a CICS region failure, a CFDT server failure, and an MVS failure (updates made by units of work that were in-flight at the time of the failure are backed out).

The recoverable locking model supports in-doubt and backout failures: if a unit of work fails when backing out an update to the CFDT, or if it fails in-doubt during syncpoint processing, the locks are converted to retained locks and the unit of work is shunted.

CFDTs cannot be forward recoverable. A CFDT does not survive the loss of the CF structure in which it resides.

You specify the update model you want for each table on its file resource definition, enabling different tables to use different models.

Techniques for sharing data

This topic indicates when you should consider using a coupling facility data table by comparing, in tabular form, the various CICS techniques that you can use for different situations.

Table 18. Techniques for sharing scratchpad data

Constraints and factors	Single Region	Single MVS	Sysplex
Technique no longer recommended (too restrictive)	TWA	—	—
Recommended method for single area for each transaction	COMMAREA or channel	COMMAREA or channel	COMMAREA or channel
Existing application programs use temporary storage (TS) queues	Local TS queue	Remote TS queue	Shared TS queue
Existing programs use UMT Random insert and delete required Multiple types of data stored	UMT	Remote UMT	CFDT (contention model)

In Table 18, different techniques are considered for passing scratchpad data between phases of a transaction, where only one task is accessing the data at a time, but the data may be passed from a task in one region to a task in another.

Note that 'remote UMT' means a shared user-maintained data table that is accessed from AORs either by function shipping where necessary (that is, for update accesses) or by SDT cross-memory sharing for non-update accesses. The table shows that, within a Parallel Sysplex®, a coupling facility data table is the best solution for random insertion and deletion of data, and where multiple types of data need to be stored. Without these constraints, shared TS queues are a more appropriate choice if the application programs are already using temporary storage.

Table 19. Techniques for sharing queues of data

Constraints and factors	Single Region	Single MVS	Sysplex
Read-only at head, write-only at tail Triggering required	Local transient data (TD)	Remote TD	Remote TD
Process batches of items	TS queue or UMT	Remote TS or remote UMT	Shared TS or CFDT
Delete each item after processing. Random insert and delete required.	UMT	Remote UMT	CFDT

In Table 19, different techniques for sharing queues of data are shown, where information is stored in a specific sequence, to be processed by another application program or task in the same sequence. The CICS transient data and temporary storage queue facilities are recommended in the majority of cases, with a few instances where data tables provide a more appropriate solution for handling sequenced data.

Table 20. Techniques for sharing control records

Constraints and factors	Single Region	Single MVS	Sysplex
Technique no longer recommended	CWA	MVS CSA	—
Single updating region, single record	TS queue or UMT	Remote TS queue or UMT	Shared TS queue or CFDT (contention model)
Multiple updating regions or multiple records	UMT	Remote UMT	CFDT

In Table 20, different techniques for managing control records are shown. This illustrates where a central control record is used to make information available to all transactions. For example, this may contain the next unused order number, or customer number, to make it easier for programs to create new records in a keyed file or database. (For this type of application, you should also consider the named counter function, which is also a sysplex-wide facility. See Chapter 25, "Named counter servers," on page 341 for details.)

The table shows that within an MVS image, if there is a single region that makes all the updates to a single record, you can use a UMT without any function shipping overheads.

Where there are multiple regions updating the control record, or there is more than one control record to update, then a coupling facility data table is the only solution within a Parallel Sysplex environment, and it could also be more effective than function shipping the updates to a UMT within a single MVS.

Table 21. Techniques for sharing keyed data

Constraints and factors	Single Region	Single MVS	Sysplex
Read-only or rarely updated	UMT	UMT	Replicated UMT
Single updating region	UMT	UMT	Replicated UMT or CFDT
Multiple updating regions Recoverable (backout only)	UMT	Remote UMT or CFDT	CFDT

In Table 21, different techniques for sharing keyed data are shown. This covers applications that use data similar in structure to a conventional keyed file, but where the information does not need to be stored permanently, and the performance benefits are sufficient to justify the use of main storage or coupling facility resources to store the relevant data.

This kind of data is most appropriately accessed using the file control API, which means that within a Parallel Sysplex, the solution is to use:

- A replicated user-maintained data table where the highest performance is required, and where access is either read-only, or updates are rare and you can arrange to make these from a single region and refresh the replicated UMT in other regions
- A coupling facility data table.

Note that recovery support for UMTs is limited to transaction backout after a failure. For coupling facility data tables, recovery is also provided for CICS and CFDT server failures, and also for in-doubt failures,

Using CICS commands to read records

This section describes the facilities available to application programs for accessing data sets. Although VSAM data sets, are discussed, most of the facilities apply equally to BDAM. It describes:

- “Direct reading (using READ command)” on page 297
- “Sequential reading (browsing)” on page 298
- “Skip-sequential processing” on page 300

A file can be defined in the file definition as containing either fixed-length or variable-length records. Fixed-length records should be defined only if:

- The definition of the VSAM data set (using access method services) specifies an average record size that is equal to the maximum record size
- **and** All the records in the data set are of that length.

For direct reading and browsing, if the file contains fixed-length records, and if the application program provides an area into which the record is to be read, that area must be of the defined length. If the file contains variable-length records, the command must also specify the length of the area provided to hold them (which should normally be the maximum length of records in the file).

For fixed-length records and for records retrieved into storage provided by CICS (when you use the SET option), you need not specify the LENGTH argument. However, although the LENGTH argument is optional, you are recommended to specify it when using the INTO option, because it causes CICS to check that the

record being read is not too long for the available data area. If you specify LENGTH, CICS uses the LENGTH field to return the actual length of the record retrieved.

Direct reading (using READ command)

You read a record in the file with the READ command. This must identify the record you want and say whether it is to be read into an area of storage provided by your application program (READ INTO), or into CICS SET storage acquired by file control (READ SET). If the latter, the address of the data in the CICS SET storage is returned to your program.

CICS SET storage normally remains valid until the next syncpoint, or the end of the task, or until next READ against the same file, whichever comes first.

Direct reading from a KSDS

When reading from a KSDS, you can identify the record you want uniquely by specifying its full key, or you can retrieve the first (lowest key) record whose key meets certain requirements. There are two options that qualify your key value; GENERIC and GTEQ.

The GENERIC option indicates that you require a match on only a part of the key; when you specify the GENERIC option, you also must specify the KEYLENGTH option, to say how many positions of the key, starting from the left, must match. For the READ command, CICS uses only the first KEYLENGTH option characters.

The GTEQ option indicates that you want the first record whose key is “greater than or equal to” the key you have specified. You can use GTEQ with either a full or a generic key.

The opposite of the GTEQ option is the EQUAL option (the default), which means that you want only a record whose key matches exactly that portion (full or generic) of the key that you have specified.

Direct reading from an ESDS

When reading from an ESDS, the individual record you want is identified by an RBA. Because the RBA of a record in an ESDS cannot change, your application program can keep track of the values of the RBAs corresponding to the records it wants to access. An RBA must always point to the beginning of a record. There is no equivalent to the GENERIC or GTEQ options that you can use to position approximately in a KSDS.

Direct reading from an RRDS

When reading from an RRDS, the record to be retrieved is identified by its relative record number. The application program must know the RRN values of the records it wants. There is no equivalent to the GENERIC or GTEQ options that you can use to position approximately in a KSDS.

Direct reading by way of a path

If a KSDS or an ESDS has an alternate index and an alternate index path (and an appropriate entry in the FCT), you can retrieve a record in the file by using the alternate key that you set up in the alternate index. The GENERIC option and the GTEQ (greater than or equal to) option still work in the same way as for a read from a KSDS using the primary key.

If the alternate key in a READ command is not unique, the first record in the file with that key is read and you get the DUPKEY condition. To retrieve other records with the same alternate key, you have to start a browse operation at this point.

Read integrity (in RLS mode)

CICS supports three options to control read integrity with RLS. You can specify these options on the file control API. Alternatively, if the application request does not specify any of the options (UNCOMMITTED, CONSISTENT, or REPEATABLE), the value from the file resource definition is used. These options are:

UNCOMMITTED

There is no read integrity and shared locks are not used for read requests. (See “RLS Record level locking” on page 313 for information about shared and exclusive locks.) This is the default and is the way in which file control works for files that are opened in non-RLS mode.

CONSISTENT

A request to read a record is queued if the record is being updated by another task. The read completes only when the update is complete, and the updating unit of work (UOW) relinquishes its exclusive lock. UOWs and syncpoints are discussed in “Syncpointing” on page 257.

REPEATABLE

Processing of the read request is the same as for consistent read requests. However, in this case, the reader holds on to its shared lock until syncpoint. This applies to both recoverable and non-recoverable files. This ensures that a record read in a UOW cannot be modified while the UOW makes further read requests. It is particularly useful when you issue a series of related read requests and you want to ensure that none of the records is modified before the last record is read.

Note: Specify read integrity only when an application cannot tolerate ‘stale’ data. This is because RLS uses locks to support read integrity, and as a result your applications could encounter deadlocks that do not occur in releases of CICS that do not support read integrity. This is particularly important if you define read integrity on file resource definitions. The application programs that reference these files may have been written for releases of CICS that do not support read integrity, and are not designed to deal with deadlock conditions on read-only file accesses.

If you specify either CONSISTENT or REPEATABLE, you can also specify the NOSUSPEND option on a READ command to ensure that the request does not wait if the record is locked by VSAM with an active lock. See “Active and retained states for locks” on page 315 for more information about active locks.

Sequential reading (browsing)

You start a browse with the STARTBR command, identifying a particular record in the same way as for a direct read. However, the STARTBR command only identifies the starting position for the browse; it does not retrieve a record.

The READNEXT command reads records sequentially from this starting point. On completion of each READNEXT command, CICS returns the full key of the record it retrieved in the field specified in the RIDFLD option. (Be sure to provide a field as long as the full key, even if you use a STARTBR command with a shorter generic key.)

As in the case of a direct read, the record may be read into an area supplied by the application program (when you use the INTO option), or into storage provided by CICS (when you use the SET option). In the latter case, the CICS storage addressed by the SET pointer remains valid until the next operation in the browse, or until the browse ends, syncpoint, or end of task, whichever occurs first.

You can also browse backwards in the file, by using READPREV commands instead of READNEXT commands, and you can switch from one direction to the other at any time. The READPREV command is like the READNEXT command, except that the records are read sequentially **backward** from the current position. As you switch from one direction to the other, you retrieve the same record twice, unless you reposition.

When the browse has started, you can change the current browse position either by using a RESETBR command, or a READNEXT command, or a READPREV command. The RESETBR command can also be used for other purposes, however.

For VSAM, but not for BDAM, you can reposition simply by varying the value in RIDFLD when you issue the next READNEXT or READPREV command. When you change RIDFLD, the record identifier must be in the same form as on the previous STARTBR or RESETBR command (key, RBA, or RRN). In addition, you can change the length of a generic key by specifying a KEYLENGTH in your READPREV command, which is different from the current generic key length and not equal to the full length. If you change the length of a generic key in this way, you reposition to the generic key specified by RIDFLD option.

RESETBR command must be used to change the browse position in a BDAM file. If you wish to change the form of the key from key to RBA or vice versa, you must use a RESETBR command. You must also use a RESETBR command to switch between generic and full keys or between “equal to” and “greater than or equal to” searches. You can also only use X'FF' characters to point to the last record in the file if you are using a RESETBR or STARTBR command.

Under certain conditions, CICS uses VSAM skip-sequential processing when you change the key in this way, as explained in “Skip-sequential processing” on page 300.

Browsing through a KSDS

You can use a generic key on the STARTBR command when browsing through a KSDS. However, the browse can only continue forward through the file. If you process a READPREV command during such a browse, you get the INVREQ condition.

You can use the options “key equal to” and “key greater than or equal to” on the STARTBR command and they have the same meaning as on the READ command. However, the STARTBR command assumes that you want to position at the key specified or the first key greater if that key does not exist. That is, option GTEQ is the default for the STARTBR command, whereas EQUAL is the default for the READ command.

You can start a forward browse through a KSDS at the start of the file by specifying a key of hexadecimal zeros, or by specifying options GENERIC, GTEQ, and KEYLENGTH(0) on the STARTBR, RESETBR, READNEXT, or READPREV command. (In the latter case, you need the RIDFLD keyword although its value is not used and, after the command completes, CICS is using a generic key length of one.)

You can start from the end of the data set by specifying a complete key of X'FF' characters on the STARTBR or RESETBR command. This points to the last record in the file ready for a backward browse.

A STARTBR, RESETBR, or READNEXT command having the option KEYLENGTH(0) is always treated as if KEYLENGTH(1) and a partial key of one byte of binary zeros have been specified.

Browsing through an ESDS

Positioning for a browse in an ESDS is identical to that for reading. If you want to begin reading at the beginning of the data set, use an RBA of low values (X'00'), and to begin at the end, use high values (X'FF').

Browsing through an RRDS

You can use the GTEQ option on a STARTBR command when browsing through an RRDS. It is the default, even though on a direct READ this option has no effect. A direct read command with the GTEQ option that specifies an RRN that does not exist returns the NOTFND condition, because only the EQUAL option is taken. However, a STARTBR GTEQ command using the same RRN completes successfully, and sets a pointer to the relevant position in the data set for the start of the browse. The first record in the file is identified using an RRN of 1, and the last record by high values (X'FF').

Browsing using a path

Browsing can also use an alternate index path to a KSDS or an ESDS. The browse is just like that for a KSDS, but using the alternate key. The records are retrieved in alternate key order.

If the alternate key is not unique, the DUPKEY condition is raised for each retrieval operation except the last occurrence of the duplicate key. For example, if there are three records with the same alternate key, DUPKEY is raised on the first two, but not the third. The order in which records with duplicate alternate keys are returned is the order in which they were written to the data set. This is true whether you are using a READNEXT or a READPREV command. For this reason, you cannot retrieve records with the same alternate key in reverse order.

Browse integrity (in RLS mode)

The options UNCOMMITTED, CONSISTENT, REPEATABLE, and NOSUSPEND, discussed in “Read integrity (in RLS mode)” on page 298, also apply to the CICS browse commands.

Ending the browse

Trying to browse past the last record in a file raises the ENDFILE condition. Stop a browse with the ENDBR command. You must issue the ENDBR command before performing an update operation on the same file (a READ UPDATE, DELETE with RIDFLD, or WRITE command). If you do not, you get unpredictable results, possibly including deadlock within your own transaction.

Simultaneous browse operations

CICS allows a transaction to perform more than one browse on the same file at the same time. You distinguish between browse operations by including the REQID option on each browse command.

Skip-sequential processing

When possible, CICS uses VSAM “skip-sequential” processing to speed browsing. Skip-sequential processing applies only to forward browsing of a file when RIDFLD

is specified in key form. CICS uses it when you increase the key value in RIDFLD on your READNEXT command and make no other key-related specification, such as KEYLENGTH. In this situation, VSAM locates the next desired record by reading forward through the index, rather than repositioning from scratch. This method is faster if the records you are retrieving are relatively close to each other but not necessarily adjacent; it can have the opposite effect if the records are very far apart in a large file. If you know that the key you are repositioning to is much higher in the file, and that you may incur a long index scan, you may wish to consider using a RESETBR command which forces a reposition from scratch.

Using CICS commands to update records

To update a record, you must first retrieve it using one of the file control read commands that specifies the UPDATE option. The record is identified in exactly the same way as for a direct read. In a KSDS or ESDS, the record may (as with a direct read) be accessed by way of a file definition that refers either to the base, or to a path defined to it. For files opened in RLS mode you can specify the NOSUSPEND option as well as the UPDATE option on an EXEC CICS command to ensure that the request does not wait if the record is already locked by VSAM with an active lock.

After modification by the application program, the record is written back to the data set using the REWRITE command, which does not identify the record being rewritten. Within a unit of work, each REWRITE command should be associated with a previous READ UPDATE by a common keyword (TOKEN). You can have one READ UPDATE without a TOKEN outstanding at any one time. Attempts to perform multiple concurrent update requests within a unit of work, upon the same data set without the use of TOKENS, are prevented by CICS. If you want to release the string held by a READ UPDATE without rewriting or deleting the record, use the UNLOCK command. The UNLOCK command releases any CICS storage acquired for the READ command, and releases VSAM resources held by the READ command. If TOKEN is specified with the UNLOCK command, CICS attempts to match this with an outstanding READ UPDATE whose TOKEN has the same value. (For more explanation about the TOKEN option, see “The TOKEN option” on page 306.)

For both update and non-update commands, you must identify the record to be retrieved by the record identification field specified in the RIDFLD option. Immediately on completion of a READ UPDATE command, the RIDFLD data area is available for reuse by the application program.

A record retrieved as part of a browse operation can only be updated during the browse if the file is opened in RLS mode (see “Updating and deleting records in a browse (VSAM RLS only)” on page 302). For files opened in non-RLS mode, the application program must end the browse, read the desired record with a READ UPDATE command, and perform the update. Failure to end the browse before issuing the READ UPDATE command may cause a deadlock.

The record to be updated may (as in the case of a direct read) be read into an area of storage supplied by the application program or into storage set by CICS. For a READ UPDATE command, CICS SET storage remains valid until the next REWRITE, UNLOCK, DELETE without RIDFLD, or SYNCPOINT command, whichever is encountered first.

For a KSDS, the base key in the record must not be altered when the record is modified. Similarly, if the update is being made by way of a path, the alternate key

used to identify the record must not be altered either, although other alternate keys may be altered. If the file definition allows variable-length records, the length of the record may be changed.

The length of records in an ESDS, a fixed-length RRDS, or a fixed-length KSDS must not be changed on update.

For a file defined to CICS as containing fixed-length records, the length of record being rewritten **must equal the original length**. For variable-length records, you must specify the LENGTH option with both the READ UPDATE and the REWRITE commands. The length must not be greater than the maximum defined to VSAM.

Using CICS commands to delete records

Records can never be deleted from an ESDS.

Deleting single records

You delete a single record in a KSDS or RRDS in one of three ways:

1. Retrieve it for update with a READ UPDATE command, and then issue a DELETE command without specifying the RIDFLD option.
2. Issue a DELETE command specifying the RIDFLD option.
3. For a file opened in RLS mode, retrieve the record with a READNEXT or READPREV command with the UPDATE option, and then issue a DELETE command. This method is described in “Updating and deleting records in a browse (VSAM RLS only).”

If a full key is provided with the DELETE command, a single record with that key is deleted. So, if the data set is being accessed by way of an alternate index path that allows non-unique alternate keys, only the first record with that key is deleted. After the deletion, you know whether further records exist with the same alternate key, because you get the DUPKEY condition if they do.

Updating and deleting records in a browse (VSAM RLS only)

For files accessed in RLS mode, you can specify the UPDATE option on a READNEXT or READPREV command and then update or delete the record by issuing a DELETE or REWRITE command. If the browse command returns a TOKEN, the TOKEN remains valid only until the next browse request. The TOKEN is invalidated by REWRITE, DELETE, or UNLOCK commands, that specify the same value for TOKEN or by the commands READNEXT, READPREV, or ENDBR that specify the same REQID. If you issue many READNEXT commands with the UPDATE and TOKEN options, the TOKENS invalidate each other and only the last one will be usable. (For more explanation about the TOKEN option, see “The TOKEN option” on page 306.)

Use of the UPDATE option in a browse is subject to the following rules:

- You can specify UPDATE within a browse only if the file is accessed in RLS mode. If you specify UPDATE for a file accessed in non-RLS mode, CICS returns an INVREQ condition.
- You can specify UPDATE only on the READNEXT and READPREV commands, not on the STARTBR or RESETBR commands.
- CICS supports only one TOKEN in a browse sequence, and the TOKEN value on each READNEXT or READPREV command overwrites the previous value.
- You can mix update and non-update requests within the same browse.

- You must specify on the READNEXT, DELETE, or UNLOCK command the TOKEN to be returned by the corresponding READNEXT or READPREV command.

Locks for UPDATE: Specifying UPDATE on a READNEXT or READPREV command acquires an exclusive lock. The duration of these exclusive locks within a browse depends on the action your application program takes and on whether the file is recoverable or not.

- If the file is recoverable and you decide to DELETE or REWRITE the last record acquired by a read for update in a browse (using the associated token), the VSAM exclusive lock remains active until completion of the UOW. That is, until successful syncpoint or rollback.
- If the file is not recoverable and you decide to DELETE or REWRITE the last record acquired, the lock is released either when you next issue an ENDBR command or when you issue a subsequent READNEXT or READPREV command. This is explained more fully in “RLS Record level locking” on page 313.
- If you decide *not* to update the last record read, CICS frees the exclusive lock either when your program issues another READNEXT or READPREV command in the browse, or ends the browse.

Note: An UNLOCK command does *not* free an RLS exclusive lock held by VSAM against a record acquired during a browse operation. An UNLOCK within a browse simply invalidates the TOKEN returned by the last request. Another READNEXT or READPREV in the browse also invalidates the TOKEN for the record read by the previous READNEXT or READPREV UPDATE command. Therefore, it's not necessary to use UNLOCK in an application program that decides not to update a particular record.

Deleting groups of records (generic delete)

You can use a generic key with the DELETE command. Then, instead of deleting a single record, all the records in the file whose keys match the generic key are deleted with the single command. However, this cannot be used if the KEYLENGTH value is equal to the length of the whole key (even if duplicate keys are allowed). The number of records deleted is returned to the application program if the NUMREC option is included with the command. If access is by way of an alternate index path, the records deleted are all those whose alternate keys match the generic key.

Read integrity

The NOSUSPEND option discussed in “Read integrity (in RLS mode)” on page 298, also applies to the CICS browse commands when you are using them to update a file.

Using CICS commands to add records

Add new records to a file with the WRITE command. They must always be written from an area provided by the application program.

Adding to a KSDS

When adding a record to a KSDS, the base key of the record identifies the position in the data set where the record is to be inserted. Although the key is part of the record, CICS also requires the application program to specify the key separately using the RIDFLD option on the WRITE command.

A record added to a KSDS by way of an alternate index path is also inserted into the data set in the position determined by the base key. However, the command must also include the alternate index key as the record identifier.

Adding to an ESDS

A record added to an ESDS is always added to the end of the file. You cannot insert a record in an ESDS between existing records. After the operation is completed, the relative byte address in the file where the record was placed is returned to the application program.

When adding a record to an ESDS by way of an alternate index path, the record is also placed at the end of the data set. The command must include the alternate index key in the same way as for a KSDS path.

Adding to an RRDS

To add a record to an RRDS, include the relative record number as a record identifier on the WRITE command. The record is then stored in the file in the position corresponding to the RRN.

Records that are already locked

The NOSUSPEND option, described in “Read integrity (in RLS mode)” on page 298 also applies to the WRITE command for a file opened in RLS mode.

Specifying record length

When writing to a fixed-length file, the record length must match the value specified at the time the file was created. In this case, you need not include the length with the command, although you may do so to check whether the length agrees with that originally defined to VSAM. If the file is defined as containing variable-length records, the command must always include the length of the record.

Sequential adding of records (WRITE MASSINSERT command)

MASSINSERT on a WRITE command offers potential improved performance where there is more than one record to add to a KSDS, ESDS, or path. The performance improvement is only obtained when the keys in successive WRITE MASSINSERT requests are in ascending order.

A MASSINSERT is completed by the UNLOCK command. This ensures that all the records are written to the file and the position is released. Always issue an UNLOCK command before performing an update operation on the same data set (read update, delete with RIDFLD, or write). If you do not, you may get unpredictable results, possibly including a deadlock.

Without an UNLOCK command, the MASSINSERT is completed when a syncpoint is issued, or at task termination.

Note: A READ command does not necessarily retrieve a record that has been added by an incomplete MASSINSERT operation.

See “VSAM data sets” on page 173 for more information about MASSINSERT.

File control command options

Some of the file control command options you may specify are:

- “The RIDFLD option”
- “The INTO and SET options”
- “The FROM option” on page 306
- “The TOKEN option” on page 306
- “KEYLENGTH option for remote data sets” on page 306

Use of the LENGTH option varies, depending on how you use the other options.

The RIDFLD option

Whatever you do to a record (read, add, delete, or start a browse), you identify the record by the RIDFLD option, except when you have read the record for update first. However, there is no RIDFLD for ENDBR, REWRITE, and UNLOCK commands. Further, during a browse using READNEXT or READPREV commands, you must include the RIDFLD option to give CICS a way to return the identifier of each record retrieved.

The RIDFLD option identifies a field containing the record identification appropriate to the access method and the type of file being accessed.

The RIDFLD option by itself is not always enough to identify a specific record in the file. So, when retrieving records from a KSDS, or from a KSDS or ESDS by way of an alternate index path, or when setting a starting position for a browse in this type of data set, you can have one or both of the further options GTEQ and GENERIC with your command.

With READNEXT or READPREV commands, the application program would not usually set the RIDFLD field. After each command, CICS updates this field with the actual identifier of the record retrieved. (You can alter the RIDFLD value to set a new position from which to continue the browse.)

The INTO and SET options

With the READ, READNEXT or READPREV commands, the record is retrieved and put in main storage according to your INTO and SET options.

The INTO option specifies the main storage area into which the record is to be put.

For fixed-length records, you need not include the LENGTH option. If you do, the length specified must exactly match the defined length; otherwise, you get the LENGERR condition.

For variable-length records, always specify (in the LENGTH option) the longest record your application program accepts (which must correspond to the value defined as the maximum record size when the data set was created); otherwise, you get the LENGERR condition. LENGERR occurs if the record exceeds the length specified, and the record is then truncated to that length. After the record retrieval, if you include the LENGTH option, the data area specified in it is set to the actual record length (before any truncation occurs).

The SET option specifies a pointer to the buffer in main storage acquired by CICS to hold the record. When using the SET option, you need not include the LENGTH option. If you do include it, the data area specified is set to the actual record length after the record has been retrieved.

The FROM option

When you add records (using the EXEC CICS WRITE command), or update records (using the REWRITE command), specify the record to be written with the FROM option.

The FROM option specifies the main storage area that contains the record to be written. In general, this area is part of the storage owned by your application program. With the REWRITE command, the FROM area is usually (but not necessarily) the same as the corresponding INTO area on the READ UPDATE command. The length of the record can be changed when rewriting to a KSDS with variable-length records.

Always include the LENGTH option when writing to a file with variable-length records. If the value specified exceeds the maximum allowed in the cluster definition, LENGERR is raised when the command is executed. LENGERR is also raised if the LENGTH option is omitted when accessing a file with variable-length records.

When writing to a file with fixed-length records, CICS uses the length specified in the cluster definition as the length of the record to be written, so you need not have the LENGTH option. If you do, its value is checked against the defined value and you get a LENGERR condition if the values do not match.

The TOKEN option

The TOKEN option is a unique value within a task that is supplied by CICS on any valid read for update command, and you return this to CICS with an associated REWRITE, DELETE, or UNLOCK command. For each file that is being updated by a task, at any one time you can have only one outstanding read request with the UPDATE option that does not specify the TOKEN option.

You can perform multiple concurrent updates on the same data set using the same task by including the TOKEN option with a read for update command, and specifying the token on the associated REWRITE, DELETE, or the UNLOCK command. Note that, for files accessed in non-RLS mode, a set of concurrent updates fails if more than one record is being updated in the same CI, irrespective of the TOKEN associated with the request. Also, only one token remains valid for a given REQID on a browse, and that is the one returned on the last READNEXT or READPREV command (see “Updating and deleting records in a browse (VSAM RLS only)” on page 302).

You can function ship a read for update request containing the TOKEN option. However, if you function ship a request specifying TOKEN to a member of the CICS family of products that does not recognize this keyword, the request fails.

KEYLENGTH option for remote data sets

In general, file control commands need the RIDFLD and KEYLENGTH options. The KEYLENGTH option can be specified explicitly in the command, or determined implicitly from the file definition.

For remote files for which the SYSID option has been specified, the KEYLENGTH option must be specified if the RIDFLD option is passing a key to file control. If the remote file is being browsed, the KEYLENGTH option is not required for the READNEXT or READPREV command.

For a remote BDAM file, where the DEBKEY or DEBREC options have been specified, KEYLENGTH (when specified explicitly) should be the total length of the key (that is, all specified subfields).

Transaction deadlocks

Design your applications so as to avoid transaction deadlocks. A deadlock occurs if each of two transactions (for example, A and B) needs exclusive use of some resource (for example, a particular record in a data set) that the other already holds. Transaction A waits for the resource to become available. However, if transaction B is not in a position to release it because it, in turn, is waiting on some resource held by A, both are deadlocked and the only way of breaking the deadlock is to cancel one of the transactions, thus releasing its resources.

A transaction needs exclusive control of resources while executing file control commands. For both VSAM and BDAM data sets, any record that is being modified is held in exclusive control by CICS from the time when the modification begins (for example, when a READ UPDATE command is issued to obtain control of the record), to the time when it ends (for example, when a REWRITE command has finished making a change to the record).

With VSAM files accessed in RLS mode, only the individual record is ever locked during this process. With VSAM files accessed in non-RLS mode, when VSAM receives a command that requires control of the record, it locks the complete control interval containing the record. CICS then obtains an enqueue on the record that it requires, and releases the control interval, leaving only the record locked. The control interval lock is released after each command, and only the individual record is locked for the whole of the modification process. (For more information about how the control interval lock is released, see *CICS Recovery and Restart Guide*.)

In releases prior to CICS Transaction Server for z/OS, Version 2 Release 2, the access method would also hold its lock on the complete control interval between the commands, from the time when the modification began, to the time when it ended. This is no longer the case.

As well as CICS having exclusive control of a record during the modification process, there is an extra locking period when a transaction modifies a record in a **recoverable** file. In this situation, CICS (or VSAM if the file is accessed in RLS mode) locks that record to the transaction even after the request that performed the change has completed. The transaction can continue to access and modify the same record; other transactions must wait until the transaction releases the lock, either by terminating or by issuing a syncpoint request. For more information, see “Syncpointing” on page 257.

Whether a deadlock actually occurs depends on the relative timing of the acquisition and release of the resources by different concurrent transactions. Application programs may continue to be used for some time before meeting circumstances that cause a deadlock; it is important to recognize and allow for the possibility of deadlock early in the application program design stages.

Here are examples of different types of deadlock found in recoverable data sets:

- Two transactions running concurrently are modifying records within a single recoverable file, through the same FCT entry, as follows:

Transaction 1
 READ UPDATE record 1
 UNLOCK record 1

 WRITE record 2

Transaction 2
 DELETE record 2

 READ UPDATE record 1
 REWRITE record 1

Transaction 1 has acquired the record lock for record 1 (even though it has completed the READ UPDATE command with an UNLOCK command). Transaction 2 has similarly acquired the record lock for record 2. The transactions are then deadlocked because each wants to acquire the CICS lock held by the other. The CICS lock is not released until syncpoint.

- Two transactions running concurrently are modifying two recoverable files as follows:

Transaction 1		Transaction 2	
READ UPDATE	file 1, record 1	READ UPDATE	file 2, record 2
REWRITE	file 1, record 1	REWRITE	file 2, record 2
READ UPDATE	file 2, record 2	READ UPDATE	file 1, record 1
REWRITE	file 2, record 2	REWRITE	file 1, record 1

Here, the record locks have been acquired on different files as well as different records; however, the deadlock is similar to the first example.

For VSAM files accessed in non-RLS mode, CICS detects deadlock situations, and a transaction about to enter a deadlock is abended with the abend code AFCF if it is deadlocked by another transaction, or with abend code AFCG if it has deadlocked itself.

VSAM-detected deadlocks (RLS only)

With files accessed in RLS mode, deadlocks can arise between two different CICS regions, possibly running under different MVS images. In these cases, deadlock detection and resolution cannot be performed by CICS, and therefore it is performed by VSAM.

If VSAM detects an RLS deadlock condition it returns a deadlock exception condition to CICS, causing CICS file control to abend the transaction with an AFCW abend code. CICS also writes messages and trace entries that identify the members of the deadlock chain.

However, VSAM cannot detect a cross-resource deadlock (for example, a deadlock arising from use of RLS and DB2 resources) where another resource manager is involved. A cross-resource deadlock is resolved by VSAM when the timeout period expires, and the waiting request is timed out. In this situation, VSAM cannot determine whether the timeout is caused by a cross-resource deadlock, or a timeout caused by another transaction acquiring an RLS lock and not releasing it. In the event of a timeout, CICS writes trace entries and messages to identify the holder of the lock for which a timed-out transaction is waiting.

All file control requests issued in RLS mode have an associated timeout value. This timeout value is that defined by DTIMEOUT if DTIMEOUT is active for the transaction, or FTIMEOUT from the system initialization table if DTIMEOUT is not active.

Rules for avoiding deadlocks

You can avoid deadlocks by following these rules:

- All applications that update (modify) multiple resources should do so in the same order. For instance, if a transaction is updating more than one record in a data set, it can do so in ascending key order. A transaction that is accessing more than one file should always do so in the same predefined sequence of files.

If a data set has an alternate index, beware of mixing transactions that perform several updates by the base key with transactions that perform several updates by the alternate key. Assume that the transactions that perform updates always access records in ascending key sequence. Then transactions that perform all updates by the base key will not deadlock with other transactions that perform all updates by the base key. Likewise, transactions that perform all updates by the alternate key do not deadlock with other transactions that perform all updates by the alternate key. But transactions that perform all updates by the base key may deadlock with transactions that perform all updates by the alternate key. This is because the key that is locked is always the base key. Consequently, a transaction performing updates by the alternate key may be acquiring locks in a different order to a transaction performing updates by the base key.

- An application that issues a READ UPDATE command should follow it with a REWRITE, DELETE without RIDFLD, or UNLOCK command to release the position before doing anything else to the file, or should include the TOKEN option with both parts of each update request.
- A sequence of WRITE MASSINSERT commands must terminate with the UNLOCK command to release the position. No other operation on the file should be performed before the UNLOCK command has been issued.
- An application must end all browses on a file by means of ENDBR commands (thereby releasing the VSAM lock) before issuing a READ UPDATE, WRITE, or DELETE with RIDFLD command, to the file.

Chapter 21. File control—VSAM considerations

This chapter explains how to perform:

- “VSAM record identification”
- “Locking of VSAM records in recoverable files” on page 312
- “RLS Record level locking” on page 313
- “Active and retained states for locks” on page 315
- “CICS locking for writing to ESDS” on page 316

VSAM record identification

You identify records in data sets by:

- Key
- Relative byte address (RBA) and relative record number (RRN)

Key

Generally, if you use a key, you can specify either a complete key or a generic (partial) key. The exceptions to this rule are when you write a record to a KSDS or when you write a record by an alternate index path. In either of these cases you **must** specify the complete key in the RIDFLD option of the command.

When you use a generic key, you must specify its length in the KEYLENGTH option and you must specify the GENERIC option on the command. A generic key cannot have a key length equal to the full key length. You must define it to be shorter than the complete key.

You can also specify the GTEQ option on certain commands, for both complete and generic keys. The command then positions at, or applies to, the record with the next higher key if a matching key cannot be found. When accessing a data set by way of an alternate index path, the record identified is the one with the next higher alternate key when a matching record cannot be found.

Even when using generic keys, always use a storage area for the record identification field that is equal in length to the length of the complete key. During a browse operation, after retrieving a record, CICS copies into the record identification area the actual identifier of the record retrieved. CICS returns a complete key to your application, even when you specified a generic key on the command. For example, a generic browse through a KSDS returns the complete key to your application on each READNEXT and READPREV command.

Relative byte address (RBA) and relative record number (RRN)

You can use the RBA and RRN options on most commands that access data sets. In effect, they define the format of the record identification field (RIDFLD). Unless you specify either the RBA or the RRN, the RIDFLD option should hold a key to be used for accessing a KSDS (or a KSDS or ESDS by way of an alternate index).

RBA

RBA specifies that the record identification field contains the relative byte address of the record to be accessed. A relative byte address is used to access an ESDS, and it may also be used to access a KSDS that is not opened in RLS access mode. All file control commands that refer to an ESDS base, and specify the RIDFLD option, must also specify the RBA option.

Note: If a KSDS is accessed in this way, the RBA of the record may change during the transaction as a result of another transaction adding records to, or deleting records from, the same data set.

RRN

RRN specifies that the record identification field contains the relative record number of the record to be accessed. The first record in the data set is number one. All file control commands that refer to an RRDS, and specify the RIDFLD option, must also specify the RRN option.

Locking of VSAM records in recoverable files

Earlier, the prevention of transaction deadlocks in terms of the record locks acquired whenever records in a recoverable file are modified was explained. These locks are acquired by VSAM if the file is accessed in record-level sharing (RLS) mode, and by CICS if not. The locks are held on behalf of the transaction doing the change until it issues a syncpoint request or terminates (at which time a syncpoint is automatically performed). For VSAM recoverable file processing, note the following:

- Whenever a VSAM record is obtained for modification or deletion, CICS file control (or VSAM in the case of RLS) locks the record with an ENQUEUE request using the primary record identifier as the enqueue argument.

If a record is modified by way of a path, the enqueue uses the base key or the base RBA as an argument. So CICS permits only one transaction at a time to perform its request, the other transactions having to wait until the first has reached a syncpoint.

- For the READ UPDATE and REWRITE-related commands the record lock is acquired as soon as the READ UPDATE command has been issued.

For a DELETE command that has not been preceded by a READ UPDATE command, or for a WRITE command, the record lock is acquired at the time the VSAM command is executed.

For a WRITE MASSINSERT command (which consists of a series of WRITE commands), a separate record lock is acquired at the time each individual WRITE command is performed. Similarly, for a DELETE GENERIC command, each record deleted acquires a separate lock on behalf of the transaction issuing the request.

Update locks and delete locks (non-RLS mode only)

The record locks referred to above are known as update locks, because they are acquired whenever a record is updated (modified). A further type of lock (a delete lock) may also be acquired by file control whenever a DELETE, WRITE, or WRITE MASSINSERT command is being performed for a recoverable KSDS or a recoverable path over a KSDS. A delete operation therefore may acquire two separate locks on the record being deleted.

The separate delete lock is needed because of the way file control does its write operations. Before executing a WRITE MASSINSERT command to a KSDS or RRDS, file control finds and locks the empty range into which the new record or records are to go. The empty range is locked by identifying the next existing record in the data set and acquiring its delete lock.

The empty range is locked to stop other requests simultaneously adding records into it. Moreover, the record defining the end of the empty range cannot be removed during the add operation. If another transaction issues a request to add records into

the empty range or to delete the record at the end of the range, the delete lock makes the transaction wait until the WRITE or WRITE MASSINSERT command is complete. The record held with a delete lock may, however, be **updated** by another transaction during the write operation if it is in another CI.

Unlike an update lock, a delete lock is held only for the duration of a delete or write operation, or a sequence of WRITE MASSINSERT commands terminated by an UNLOCK command. A WRITE MASSINSERT command that adds records to the file into more than one empty range releases the previous delete lock as it moves into a new empty range.

The CICS enqueueing mechanism is only for updates and deletes and does not prevent a read request being satisfied before the next syncpoint. The integrity of a READ command in these circumstances is not guaranteed.

RLS Record level locking

Files opened in RLS mode can be accessed by many CICS regions simultaneously. This means it is impractical for the individual CICS regions to attempt to control record locking, and therefore VSAM maintains a single central lock structure using the lock-assist mechanism of the MVS coupling facility. This central lock structure provides sysplex-wide locking at a record level—control interval (CI) locking is not used. This is in contrast to the locks for files in non-RLS mode, the scope of which is limited to a single CICS region, and that are either CI locks or CICS ENQs.

Record locks within RLS are owned by a named UOW within a named CICS region. The lock owner name is the APPLID of the CICS region, plus the UOW id. For example, when CICS makes a request that may create a lock, CICS passes to VSAM the UOW id. This enables VSAM to build the lock name using the UOW id, the record key, and the name of the CICS region.

CICS releases all locks on completion of a UOW using a VSAM interface.

When more than one request requires an exclusive lock against the same resource, VSAM queues the second and subsequent requests until the resource is freed and the lock can be granted. However, VSAM does not queue requests for resources locked by a retained lock (see “Active and retained states for locks” on page 315).

Note: For MASSINSERT operations on a file opened in RLS access mode, CICS acquires a separate update lock at the time each individual WRITE command is issued. Unlike the non-RLS mode operation (described under “Locking of VSAM records in recoverable files” on page 312) CICS does *not* acquire the separate delete lock in addition to the update lock. There is no equivalent to range locking for the MASSINSERT function for files opened in non-RLS mode.

Exclusive locks and shared locks

VSAM supports two types of lock for files accessed in RLS mode:

1. Exclusive locks
2. Shared locks

Exclusive locks can be active or retained; shared locks can only be active (see “Active and retained states for locks” on page 315). Note that there are no delete locks in RLS mode.

Exclusive locks

Exclusive locks protect updates to file resources, both recoverable and non-recoverable. They can be owned by only one transaction at a time. Any transaction that requires an exclusive lock must wait if another task currently owns an exclusive lock or a shared lock against the requested resource.

Shared locks

Shared locks support read integrity (see “Read integrity (in RLS mode)” on page 298). They ensure that a record is not in the process of being updated during a read-only request. Shared locks can also be used to prevent updates of a record between the time that a record is read and the next syncpoint.

A shared lock on a resource can be owned by several tasks at the same time. However, although several tasks can own shared locks, there are some circumstances in which tasks can be forced to wait for a lock:

- A request for a shared lock must wait if another task currently owns an exclusive lock on the resource.
- A request for an exclusive lock must wait if other tasks currently own shared locks on this resource.
- A new request for a shared lock must wait if another task is waiting for an exclusive lock on a resource that already has a shared lock.

Lock duration

Shared locks for repeatable read requests, for recoverable and non-recoverable data sets, are held until the next syncpoint.

Exclusive locks against records in a non-recoverable data set remain held only for the duration of the request—that is, they are acquired at the start of a request and released on completion of it. For example, a lock acquired by a WRITE request is released when the WRITE request is completed, and a lock acquired by a READ UPDATE request is released as soon as the following REWRITE or DELETE request is completed. Exceptionally, locks acquired by sequential requests may persist beyond the completion of the immediate operation. Sequential requests are WRITE commands that specify the MASSINSERT option and browse for update requests. A lock acquired by a WRITE command with the MASSINSERT option is always released by the time the corresponding UNLOCK command completes, but may have been released by an earlier request in the WRITE MASSINSERT sequence. The exact request in the sequence that causes the lock to be released is not predictable. Similarly, a lock acquired by a READNEXT UPDATE command may still exist after the following DELETE or REWRITE command completes. Although this lock is guaranteed to be released by the time the subsequent ENDBR command completes, it may be released by some intermediate request in the browse sequence.

If a request is made to update a recoverable data set, the associated exclusive lock must remain held until the next syncpoint. This ensures that the resource remains protected until a decision is made to commit or back out the request. If CICS fails, VSAM continues to hold the lock until CICS is restarted.

Task 1	Task 2
CICS: READ(filea) UPDATE KEY(99)	
VSAM: grants exclusive lock - key 99	
	CICS: READ(filea) KEY(99) with integrity
	VSAM: Queues request for shared lock
CICS: REWRITE(filea) KEY(99)	
VSAM: holds exclusive lock until syncpoint	
CICS: task completes and takes syncpoint	
VSAM: frees exclusive lock	
	VSAM grants shared lock to task 2

Figure 79. Illustration of lock contention between CICS tasks on a recoverable data set

Active and retained states for locks

VSAM RLS supports **active** and **retained** states for locks. The difference between these two types of lock is that whereas a request for a resource that has an active lock is queued until the resource becomes available, a request for a resource that has a retained lock fails immediately.

The active state is applicable to both exclusive and shared locks. However, only exclusive locks against recoverable resources can have their state changed from active to retained. The important characteristic of these states is that they determine whether or not a task must wait for a lock:

- A request for a lock is made to *wait* if there is already an active lock against the requested resource, except in two cases:
 1. A request for a shared lock does not have to wait if the current active lock is also a shared lock, and there are no exclusive lock requests waiting.
 2. An update request that specifies NOSUSPEND does not wait for a lock if an active lock already exists. In this case, CICS returns an exception condition indicating that the “record is busy”.
- A request for a lock is rejected immediately with the LOCKED condition if there is a retained lock against the requested resource.

When a lock is first acquired, it is an active lock. It is then either released, the duration of the lock depending on the type of lock, or if an event occurs which causes a UOW to fail temporarily and which would therefore cause the lock to be held for an abnormally long time, it is converted into a retained lock. The types of event that can cause a lock to become retained are:

- Failure of the CICS system, the VSAM server or the whole MVS system
- A unit of work entering the backout failed state
- A distributed unit of work becoming indoubt owing to the failure of either the coordinating system or of links to the coordinating system

If a UOW fails, VSAM continues to hold the exclusive record locks that were owned by the failed UOW for recoverable data sets. To avoid new requests being made to wait for locks owned by the failed UOW, VSAM converts the active locks owned by the failed UOW into retained locks. Retaining locks ensures that data integrity for the locked records is maintained until the UOW is completed.

Exclusive recoverable locks are also converted into retained locks in the event of a CICS failure, to ensure data integrity is maintained until CICS is restarted and performs recovery.

Exclusive recoverable locks are also converted into retained locks if the VSAM data-sharing server (SMSVSAM) fails (the conversion is carried out by the other servers in the Sysplex, or by the first server to restart if all servers have failed). This means that a UOW does not itself have to fail in order to hold retained RLS locks.

Any shared locks owned by a failed CICS region are discarded, and therefore an active shared lock can never become retained. Similarly, active exclusive non-recoverable locks are discarded. Only locks that are both exclusive and apply to recoverable resources are eligible to become retained.

Conditional VSAM file update requests

On file control update requests against files opened in RLS mode, you can avoid waiting for a lock by making your request conditional upon being given a lock immediately. You do this by specifying the NOSUSPEND option on the request. If another task already holds an active lock, CICS returns the RECORDBUSY condition instead of queueing your request.

You can specify NOSUSPEND on READ, READNEXT, READPREV, WRITE, REWRITE, and DELETE commands.

It is important to distinguish between the LOCKED and RECORDBUSY responses:

- A LOCKED response occurs when a request attempts to access a record that is locked by a *retained* lock.
- A RECORDBUSY response occurs when a request attempts to access a record that is locked by an active lock. Remember that this could be caused by a DEADLOCK, in which case retries may not work. It may be necessary to issue a SYNCPOINT with or without rollback to resolve the condition.

Note: Requests that specify NOSUSPEND wait for at least 1 second before CICS returns the RECORDBUSY response.

If you do not specify NOSUSPEND on your request, CICS causes it to wait for a lock if the record is already locked by an active lock. If you specify NOSUSPEND, your request receives a RECORDBUSY response if the record is locked by an active lock.

If you issue a request (with or without the NOSUSPEND option) against a record locked by a retained lock, CICS returns a LOCKED response.

File control implementation of NOSUSPEND

There is a slight difference in the way that NOSUSPEND works on file control commands compared with the way that NOSUSPEND works on other CICS commands. If you issue HANDLE CONDITION(RECORDBUSY) it does not cause NOSUSPEND to be assumed on subsequent file control requests. On the other hand, specifying HANDLE CONDITION(QBUSY) causes NOSUSPEND to be assumed on subsequent transient data commands even when it is not explicitly specified.

CICS locking for writing to ESDS

CICS write operations to ESDS are single threaded, for both RLS and non-RLS mode access. However, the lock held for serialization can be held for slightly longer for RLS-mode access compared with non-RLS mode. You can compensate for the possible increase in overhead by increasing the task priority of those transactions

that add new records to ESDS files. It is possible that when you switch an ESDS RLS mode from non-RLS mode that you might see an increase in time-outs for those transactions that add new records.

Using RLS mode access for an ESDS can also cause availability problems. If a CICS region fails while writing to an ESDS, the data set might be locked until the CICS region is restarted. It is recommended that you do not use RLS mode access for ESDS.

Chapter 22. File control—BDAM considerations

This chapter explains how to perform the following functions :

- “Identifying BDAM records”
- “Browsing records from BDAM data sets” on page 320
- “Adding records to BDAM data sets” on page 321

Note: When a blocked record is read for update, CICS maintains exclusive control of the containing block. An attempt to read a second record from the block before the first is updated (by a REWRITE command), or before exclusive control is released (by an UNLOCK command), causes a deadlock.

Identifying BDAM records

You identify records in BDAM data sets by a **block reference**, a **physical key** (keyed data set), or a **deblocking argument** (blocked-data set). The record identification (in the RIDFLD option) has a subfield for each item. These subfields, when used, must be in the above order.

Note: When using EDF, only the first of the above three fields (the block reference subfield) is displayed.

Block reference subfield

This is one of the following:

- Relative block address: 3-byte binary, beginning at relative block zero (RELTYPE=BLK).
- Relative track and record (hexadecimal format): 2-byte TT, 1-byte R (RELTYPE=HEX).

The 2-byte TT begins at relative track zero. The 1-byte R begins at relative record one.

- Relative track and record (zoned decimal format): 6-byte TTTTTT, 2-byte RR (RELTYPE=DEC).
- Actual (absolute) address: 8-byte MBBCCHHR (RELTYPE operand omitted).

The system programmer must specify the type of block reference you are using in the RELTYPE operand of the DFHFCT TYPE=FILE system macro that defines the data set.

Physical key subfield

You only need this if the data set has been defined to contain recorded keys. If used, it must immediately follow the block reference. Its length must match the length specified in the BLKKEYL operand of the DFHFCT TYPE=FILE system macro that defines the data set.

Deblocking argument subfield

You only need this if you want to retrieve specific records from a block. If used, it must immediately follow the physical key (if present) or the block reference. If you omit it, you retrieve an entire block.

The deblocking argument can be a key or a relative record number. If it is a key, specify the DEBKEY option on a READ or STARTBR command and make sure its length matches that specified in the KEYLEN operand of the DFHFCT TYPE=FILE

system macro. If it is a relative record number, specify the DEBREC option on a READ or STARTBR command. It is a 1-byte binary number (first record=zero).

Figure 80 shows examples of the following possible forms of BDAM record identifiers. The examples assume a physical key of four bytes and a deblocking key of three bytes:

- Relative block number followed by relative record number for search by relative block and deblock by relative record
- Relative block number followed by a key for search by relative block and deblock by key
- TTR followed by physical key and deblocking key for search by relative track and record and key and deblock by key
- MBBCCHHR followed by relative record number for search by actual address and deblock by relative relative record
- TTTTTRR followed by physical key and deblocking key for search by zoned decimal relative track and record and key and deblock by key
- TR followed by physical key for search by relative track and record and deblock by key

Byte Number																	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		
RELBLK#			N													Search by relative block; deblock by relative record	
RELBLK#			KEY													Search by relative block; deblock by key	
T	T	R	PH-KEY				KEY								Search by relative track and record and key; deblock by key		
M	B	B	C	C	H	H	R	N								Search by actual address; deblock by relative record	
T	T	T	T	T	T	R	R	PH-KEY				KEY				Search by zoned decimal relative track and record and key; deblock by key	
T	T	R	KEY													Search by relative track and record; deblock by key	

Figure 80. Examples of BDAM record identification

Browsing records from BDAM data sets

The record identification field must contain a block reference (for example, TTR or MBBCCHHR) that conforms to the addressing method defined for the data set. Processing begins with the specified block and continues with each subsequent block until you end the browse.

If the data set contains blocked records, processing begins at the first record of the first block and continues with each subsequent record, regardless of the contents of the record identification field.

In other words, CICS uses only the information held in the TTR or MBBCCHHR subfield of the RIDFLD to identify the record. It ignores all other information, such as physical key and relative record, or logical key.

After the READNEXT command, CICS updates the RIDFLD with the complete identification of the record retrieved. For example, assume a browse is to be started with the first record of a blocked, keyed data set, and deblocking by logical key is to be performed. Before issuing the STARTBR command, put the TTR (assuming that is the addressing method) of the first block in the record identification field. After the first READNEXT command, the record identification field might contain X'0000010504', where X'000001' represents the TTR value, X'05' represents the block key, (of length 1), and X'04' represents the logical record key.

Now assume that a blocked, nonkeyed data set is being browsed using relative record deblocking and the second record from the second physical block on the third relative track is read by a READNEXT command. Upon return to the application program, the record identification field contains X'00020201', where X'0002' represents the track, X'02' represents the block, and X'01' represents the number of the record in the block relative to zero.

Note: Specify the options DEBREC and DEBKEY on the STARTBR command when browsing blocked-data sets. This enables CICS to return the correct contents in the RIDFLD. Specifying DEBREC on the STARTBR command causes the relative record number to be returned. Specifying DEBKEY on the STARTBR command causes the logical record key to be returned.

Do not omit DEBREC or DEBKEY when browsing a blocked file. If you do, the logical record is retrieved from the block, the length parameter is set equal to the logical record length, *but* the RIDFLD is not updated with the full identification of the record. ***Do not use this method.***

Compare this with what happens if you omit the DEBREC or DEBKEY option when reading from a blocked BDAM data set. In this case, you retrieve the *whole* block, and the length parameter is set equal to the length of the block.

Adding records to BDAM data sets

When adding records to a BDAM data set, bear in mind the following:

- When adding undefined or variable-length records (keyed or nonkeyed), you must specify the track on which each new record is to be added. If space is available on the track, the record is written following the last previously written record, and the record number is put in the “R” portion of the record identification field of the record. The track specification may be in any format except relative block. If you use zoned-decimal relative format, the record number is returned as a 2-byte zoned decimal number in the seventh and eighth positions of the record identification field.

The extended search option allows the record to be added to another track if no space is available on the specified track. The location at which the record is added is returned to the application program in the record identification field being used.

When adding records of undefined length, use the LENGTH option to specify the length of the record. When an undefined record is retrieved, the application program must find out its length.

- When adding keyed fixed-length records, you must first format the data set with dummy records or “slots” into which the records may be added. You signify a dummy record by a key of X'FF's. The first byte of data contains the record number.
- When adding nonkeyed fixed-length records, give the block reference in the record identification field. The new records are written in the location specified, destroying the previous contents of that location.
- When adding keyed fixed-length records, track information only is used to search for a dummy key and record, which, when found, is replaced by the new key and record. The location of the new record is returned to the application program in the block reference subfield of the record identification field.

For example, for a record with the following identification field:

```
0 3 0  ALPHA
T T R  KEY
```

the search starts at relative track three. When control is returned to the application program, the record identification field is:

```
0 4 6  ALPHA
```

showing that the record is now record six on relative track four.

- When adding variable-length blocked records you must include a 4-byte record description field (RDF) in each record. The first two bytes specify the length of the record (including the 4-byte RDF); the other two bytes consist of zeros.

Updating records from BDAM data sets

You cannot change the record length of a variable blocked or unblocked BDAM record on a REWRITE command which specifies deblocking. You cannot change the record length of an undefined format BDAM record on a REWRITE command either.

Chapter 23. Database control

This chapter introduces DL/I databases and “DB2 databases.”

DL/I databases

You get a logical view of the database in terms of a hierarchy of segments. DL/I lets you manipulate these segments without needing to know how they are organized. DL/I databases are processed by the IBM licensed program Information Management System/Enterprise Systems Architecture (IMS/ESA®).

CICS has two programming interfaces to DL/I. We recommend that you use the higher-level EXEC DLI interface. It is straightforward, works with EDF, and can fully exploit a 31-bit environment. The lower-level DL/I programming interface is the DL/I CALL interface.

DB2 databases

DB2 databases also provide data independence, offering a logical view of the database as a series of tables that you can interrelate more or less as you wish. DB2 lets you manipulate these tables without needing to know how they are organized. DB2 databases are processed by the IBM licensed program DB2 Universal Database™ for z/OS and OS/390.

CICS has one interface to DB2—the EXEC SQL interface, which offers powerful statements for manipulating sets of tables—thus relieving the application program of record-by-record (segment-by-segment, in the case of DL/I) processing.

CICS applications that process DB2 tables can also access DL/I databases. Any CICS resources (files, transient data, and so on), DL/I, and DB2 can be accessed from within one transaction. See the *CICS IMS Database Control Guide* for information about what databases and resources you can access.

For information about SQL commands, which are not discussed in this book, see the *DB2 Universal Database for OS/390 and z/OS Application Programming and SQL Guide*.

Requests to DB2

Requests from CICS applications to DB2 are made using EXEC SQL commands. DB2 runs in its own address space, like DBCTL. The CICS-DB2 and the CICS-DBCTL interfaces are similar in that they both use the task-related user exit interface (also known as the resource manager interface (RMI)) and have a two-phase commit process. However, they differ in a number of respects. For example, the CICS-DB2 interface uses the task-related user exit interface (also known as the resource manager interface, RMI) and has a two-phase commit process. CICS supports DBCTL and remote DL/I, and has to determine at PSB schedule time which of them is being used.

When an application issues an EXEC SQL command, the following processing takes place:

1. The RMI is invoked from a stub that is link-edited to the application.
2. If this is the first DB2 request from this task, the RMI sets up a task interface element (TIE).

3. The RMI invokes the DB2 task-related user exit.
4. The DB2 task-related user exit services the request. It suspends the task that made the request: returns the response and data to the application, then resumes the requesting task, before returning control to the RMI.
5. Control is returned to the application.

The processing steps are illustrated in Figure 81. and are the responsibility of DB2, until control is returned to the RMI

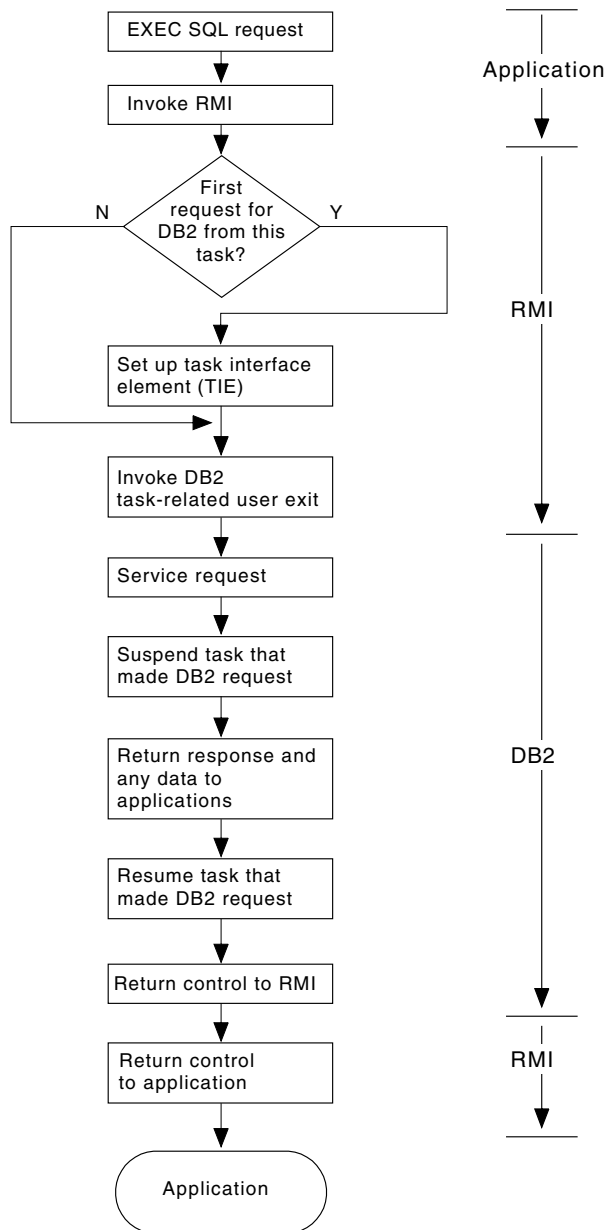


Figure 81. How EXEC SQL requests are processed

Chapter 24. CICS documents

The CICS document handler allows you to build up formatted data areas, known as documents. Some examples of how these formatted areas, or documents, can be used, are:

- Constructing a COMMAREA.
- Sending HTML data to be displayed by a Web client.
- Creating standard formats for printing (for example, using your own letterhead, address, and so on).

Using the DOCUMENT programming interface

This section explains the function and use of the commands in the DOCUMENT application programming interface:

- DOCUMENT CREATE
- DOCUMENT INSERT
- DOCUMENT RETRIEVE
- DOCUMENT SET

Creating a document

You can create an empty document using the DOCUMENT CREATE command, and then build the contents with subsequent DOCUMENT INSERT commands, or use DOCUMENT CREATE to create and build the document in one step. DOCUMENT CREATE has a mandatory DOCTOKEN parameter requiring a 16-byte data-area. The document handler domain uses the DOCTOKEN operand to return a token, which is used to identify the document on subsequent calls. The following example creates an empty document, and returns the token in the variable MYDOC:

```
EXEC CICS DOCUMENT CREATE
      DOCTOKEN(MYDOC)
```

To create a document with data, use the DOCUMENT CREATE command in any of the following ways:

- Using the BINARY option
- Using the TEXT option
- Using the FROMDOC option to copy an existing document
- Using the TEMPLATE option. “Setting up document templates” on page 326 describes this option.

The BINARY option

Use this option to add to the document the contents of a data-area that must not undergo conversion to a client code page when the data is sent.

```
EXEC CICS DOCUMENT CREATE
      DOCTOKEN(MYDOC1)
      BINARY(DATA-AREA)
```

The TEXT option

Use this option to add the specified contents to the document. For example, if you define a character string variable called DOCTEXT and initialise it to *This is an example of text to be added to a document*, you can use the following command to create a document consisting of this text string:

```
EXEC CICS DOCUMENT CREATE
      DOCTOKEN(MYDOC2)
      TEXT(DOCTEXT)
      LENGTH(53)
```

The FROMDOC option

To copy an existing document into a new document, you can use the DOCUMENT CREATE command with the FROMDOC option. The following example shows this:

```
EXEC CICS DOCUMENT CREATE
      DOCTOKEN(MYDOC3)
      FROMDOC(MYDOC2)
```

where MYDOC2 and MYDOC3 are 16-character variables. MYDOC2 must contain the token returned by a previous DOCUMENT CREATE command.

This results in two identical documents, each containing the text *This is an example of text to be added to a document.*

Setting up document templates

Document templates are portions of a document which can be created off-line, or in another CICS program, and inserted into the document in the application program. Document templates are CICS resources, which you define using DOCTEMPLATE definitions; the name of the template is specified in the TEMPLATENAME attribute.

Templates can contain static data, and symbols whose values are inserted into the template when you issue the DOCUMENT CREATE or DOCUMENT INSERT command. The values to be substituted are specified in the application program; they are associated with a particular document and cannot be used in a different document.

Templates can be retrieved from several different sources, to suit the way they are used in the application program; the DOCTEMPLATE definition specifies the source of the template:

- from a partitioned data set
- from a CICS program
- from a CICS file
- from a z/OS UNIX System Services HFS file
- from a temporary storage queue
- from a transient data queue
- from an exit program

If you attempt to use a template for which there is no installed DOCTEMPLATE definition, CICS attempts to retrieve the template from the partitioned data set with a DDNAME of DFHHTML.

Templates in a partitioned data set

Consider storing your templates in a partitioned data set when any of the following apply:

- You want to edit your template on-line, or import the template from an HTML editor.
- You want to use templates that have been created from BMS maps

CICS loads a copy of the template from the partitioned data set when you install the corresponding DOCTEMPLATE definition. Therefore you can modify the template in the partitioned data set while CICS is running, and reinstall the definition in order to activate the changes.

A partitioned data set used to store templates may have one of the following record formats:

- FB (fixed length blocked)
- VB (variable length blocked)
- U (unblocked)

Records may contain sequence numbers in the following cases:

- When the record format is FB, and the record length is 80; the sequence numbers must be in positions 73 through 80.
- When the record format is VB; the sequence numbers must be in positions 1 through 8.

In other cases, there must be no sequence numbers in the records. If you use sequence numbers, they must be present in all the records; do not use partially sequenced members.

Templates in a CICS program

Consider coding a CICS program to contain a template when:

- You need to minimize the overhead of retrieving the template
- The template is static; in other words, the contents of the template do not change dynamically
- The same template is used by a several applications

To code a program which contains a template:

1. Code an Assembler CSECT containing
 - a. An ENTRY statement, which denotes the start of the template
 - b. Character constants (DC statements) defining the text that you wish to include in your template.
 - c. An END statement

For example:

```
                ENTRY WKLYHDR
WKLYHDR CSECT
          DC     CL4 '<HR>'
          DC     CL29 '<H2>Weekly Status Report</H2>'
          END
```

2. Assemble and link edit the program into your CICS application program library. Note that the name you give to the program can be different from the name of the entry point.
3. Create and install a DOCTEMPLATE definition which specifies the name of the program in the Program attribute.

CICS will autoinstall the program on first reference, or you can create and install a PROGRAM definition.

Templates in CICS files, z/OS UNIX System Services HFS files, temporary storage, or transient data

Consider using one of these resources when you want to use dynamic data from an application program in a template. Which resource you use will depend upon:

- how the application program stores its data
- whether the existing data can be used directly in the template, or needs to be modified
- whether the data must be preserved after it is used in the template

In general, when a template is inserted into a document, all the data contained in the resource is used:

Temporary storage

The queue is read, in sequence, by ITEM number, and therefore all records in the queue are read, regardless of which records have been read by other applications.

Transient data

Because transient data uses a destructive read, when you insert data from a transient data queue into a template, the contents of the queue are no longer available to other applications.

CICS file

- Entry-sequenced data sets (ESDS) are read in sequence of relative byte address.
- Relative record data sets (RRDS) are read in sequence of relative record number.
- Other data sets are read in sequence of key field.

z/OS UNIX System Services HFS file

- All the data in the file is used.
- In the DOCTEMPLATE definition, you need to provide the fully-qualified name of the HFS file, which can be up to 255 characters in length.
- The CICS region must have permissions to access z/OS UNIX, and it must have permission to access the HFS directory containing the file, and the file itself. *Java Applications in CICS* explains how to achieve this.

Creating templates in exit programs

Consider using an exit program for a template when:

- You need to minimize the overhead of retrieving the template
- The contents of the template change dynamically
- The same template is used by a several applications
- You want to retrieve the contents of the template from a non-CICS resource (for example, DB2)

When an application program requests a template which is defined as being created in an exit program, CICS calls the specified program, and passes a communication area; the communication area is mapped by the following copybooks:

- DFHDHTXD (Assembler)
- DFHDHTXH (C)
- DFHDHTXL (PL/I)
- DFHDHTXO (COBOL)

The communication area contains the following fields:

dhtx_template_name_ptr

Contains a pointer to the name (up to 48 characters) of the template that is being requested.

dhtx_buffer_ptr

Contains the pointer of the CICS-supplied buffer in which the exit program returns the template.

dhtx_buffer_len

(Fullword binary.) Contains the length of the CICS-supplied buffer in which the exit program returns the template.

dhtx_message_len

(Fullword binary.) Use this field to return the length of a message that is issued when the exit program is unable to return a template. If there is no message, return a value of zero.

dhtx_message_ptr

Use this field to return the pointer of a message that explains why the exit program was unsuccessful. CICS writes this message to the CSDH transient data destination. If there is no message, return a value of zero.

dhtx_template_len

(Fullword binary.) Use this field to return the actual length of the template.

dhtx_append_crlf

Use the characters '1' (append) or '0' (do not append) to specify whether or not to add carriage return and line feed characters to the end of each line.

dhtx_return_code

(Fullword binary.) Use this field to indicate whether the exit program has successfully returned a template:

- A return code of 0 indicates that the exit has returned a template.
- A return code of 8 indicates that the exit has not returned a template. In this case, CICS raises a `TEMPLATERR` condition in the application program.

If the template to be returned is longer than `dhtx_buffer_len`, the template must be truncated to length `dhtx_buffer_len` and the exit program must set the length required in `dhtx_template_len`. The exit program is then called again with a larger buffer.

If your exit program sets a return code of 8, you can return an explanatory message, which is written to the CSDH transient data destination. Return the address and length of the message in `dhtx_message_ptr` and `dhtx_message_len` respectively. The storage which contains the message must be accessible to the caller of the exit program. For example, your exit program can issue a `GETMAIN` command to acquire storage for the message. CICS will release the storage when the task ends, unless you specify the `SHARED` option on the command.

Programming with documents and document templates

This section covers the following topics:

- “Symbols and symbol lists”
- “Setting symbol values” on page 330
- “Embedded template commands” on page 332
- “Using templates in your application” on page 333
- “The lifespan of a document” on page 334
- “Constructing a document” on page 336
- “Using Bookmarks” on page 338
- “Code page conversion for documents” on page 339

Symbols and symbol lists

Symbols represent variable data that is resolved at the time the template is added to the document - at the time the `DOCUMENT CREATE` or `DOCUMENT INSERT` is issued.

The application program needs to define values for the symbols that will be substituted when the template is used. These values can be defined on the DOCUMENT CREATE or the DOCUMENT SET commands. These commands both take a SYMBOLLIST operand which allows several symbols to be defined in a single command. You can also define individual symbols by using the SYMBOL and VALUE operands on the DOCUMENT SET command.

When you are planning your use of symbols and templates, note that:

1. When a template containing symbols has been inserted into a document, you cannot change the substituted values of those symbols in the document that is being composed. If you set different values for the symbols, the new values will be used the next time that the template is inserted into a document. Your changes will not affect the values that have already been inserted into the document.
2. If you insert a template before the symbols contained in it are set, the symbols will never be substituted. This can occur if you create a document from a template without specifying a symbol list.
3. Symbol substitution does not occur in html comments.

The support for symbols and symbol lists in the DOCUMENT application programming interface is designed to interpret data with a content type of **application/x-www-form-urlencoded**, as described in the HTML specification (at <http://www.w3.org/TR/html401/interact/forms.html#h-17.13.4>). This is the format in which data is returned when it has been entered into a form on a web page.

In CICS, the DOCUMENT application programming interface extends this specification in respect of space characters. You can use simple space characters to indicate spaces within values in a symbol list or in the VALUE option; you do not have to use a plus sign to indicate a space. However, if you want to adhere more closely to the specification, it is suggested that you do use a plus sign instead of a space character when possible. Note, though, that you **cannot** use a plus sign to indicate a space character when the UNESCAPED option is used to prevent CICS from unescaping symbol values contained in a symbol list or in the VALUE option. In these cases, you must use a space character to indicate a space, because plus signs are not converted to spaces when the UNESCAPED option is used.

Setting symbol values

You can define values for symbols using the DOCUMENT SET command or the DOCUMENT CREATE command.

The DOCUMENT CREATE and DOCUMENT SET commands both take a SYMBOLLIST operand which allows several symbols to be defined in a single command. The SYMBOLLIST operand is a character string consisting of one or more definitions with single byte separators. By default, the symbol separator is an ampersand, but you can override this by using the DELIMITER option of the DOCUMENT SET or DOCUMENT CREATE commands. A definition for a symbol consists of a name, an equals sign, and a value.

Here is an example of a symbol list:

```
mytitle=New+Authors&auth1=Halliwell+Sutcliffe&auth2=Stanley+Weyman
```

This example defines three symbols. The first symbol called mytitle has the value 'New Authors'. The second symbol called auth1 has the value 'Halliwell Sutcliffe' and the last symbol called auth2 has the value 'Stanley Weyman'. To adhere to the specification for the content type **application/x-www-form-urlencoded**, plus signs

have been used to indicate spaces within the symbol values; they will be converted to spaces when the symbol values are put into the symbol table.

The following rules apply when setting symbols using a SYMBOLLIST:

- The name of each symbol must contain only uppercase and lowercase letters, numbers and the special characters dollar ('\$'), underscore ('_'), hyphen ('-'), number sign ('#'), period('.') and at sign('@'). The name is case-sensitive, so uppercase letters are regarded as different from lowercase letters.
- The values in the symbol list can contain any characters. However, special coding is required if you need to include the following characters in symbol values in the symbol list:
 - The character that you have used as the symbol separator (which defaults to an ampersand, but can be overridden by use of the DELIMITER option).
 - The plus sign and the percent sign.

You can use the percent sign, followed by two characters that are hexadecimal digits (that is, 0–9, a-f, and A-F), to include characters such as these that have a special meaning. When the value is put into the symbol table, a percent sign and the two hexadecimal digits following it are interpreted as the EBCDIC equivalent of the single ASCII character denoted by the two digits. Some useful combinations are as follows:

- If you want a plus sign in the value in the symbol table, put %2B in the value in the symbol list.
- If you want a percent sign in the value in the symbol table, put %25 in the value in the symbol list.
- If you want an ampersand in the value in the symbol table, put %26 in the value in the symbol list.
- You can also use this coding to specify a space in the value in the symbol table—put %20 in the value in your symbol list. Alternatively, you can use the space character or the plus sign, as described below.

If you prefer not to use this special coding, you can specify the UNESCAPED option on the DOCUMENT CREATE or DOCUMENT SET command. When you specify this option, no conversion takes place, and the symbol values are put into the symbol table exactly as you entered them. The example below shows you how to do this. However, the UNESCAPED option does not allow you to include the character that you have used as the symbol separator, within a symbol value in a symbol list. If you want to use the UNESCAPED option, choose a symbol separator that will never be used within a symbol value. Alternatively, you can use the SYMBOL and VALUE options on the DOCUMENT SET command to specify symbol values that contain the character you have used as the symbol separator, because the symbol separator has no special meaning when used in the VALUE option.

- If you want to include spaces in a value, CICS allows you to use the space character. However, to adhere more closely to the specification for the content type, it is suggested that you use a plus sign instead of a space character. The plus sign is interpreted as a space when the symbol value is put into the symbol table. However, you **cannot** use a plus sign to indicate a space character when the UNESCAPED option is used to prevent CICS from unescaping symbol values contained in a symbol list or in the VALUE option. In these cases, you must use a space character to indicate a space, because plus signs are not converted to spaces when the UNESCAPED option is used.

The following example shows you how, without using special coding, you can pass symbol values to the document handler containing embedded plus signs, percent signs, and ampersands, none of which are to undergo unescape processing:

```
EXEC CICS DOCUMENT CREATE
      DOCTOKEN(ATOKEN)
      DELIMITER('!')
      SYMBOLLIST('COMPANY=BLOGGS & SON!ORDER=NUTS+BOLTS')
      LISTLENGTH(37)
      UNESCAPED
```

Here the symbol COMPANY has a value of 'BLOGGS & SON', and the symbol ORDER has a value of 'NUTS+BOLTS'. The symbol separator used in this example is '!', but it is best to use a non-printable character that does not appear in the symbol value. The use of a character other than the ampersand as the symbol separator means that an ampersand can be used in 'BLOGGS & SON'. The use of the UNESCAPED option ensures that the plus sign in 'NUTS+BOLTS' does not get converted to a space. Because the UNESCAPED option has been used, you must use a space character, rather than a plus sign, to indicate where a space is required in the symbol value, and the data no longer conforms to the specification for the content type **application/x-www-form-urlencoded**.

Depending on your application, you might find that instead of specifying the exact list length for your symbol list each time you define values for the symbols, it is more convenient to choose a permanent value for the LISTLENGTH option, which will provide a fixed list length for your symbol list. The fixed list length that you choose should be long enough to include the maximum length of symbol list that you expect to supply. However, on those occasions when the fixed list length that you have specified for your symbol list is greater than the actual length of the symbols that you supply, you might find that there are trailing spaces or unpredictable characters in the value of the last symbol in the list. You can avoid this issue by including an extra dummy symbol at the end of your symbol list, such as '&END='. Do not include this symbol in any templates or documents. Any trailing spaces or unpredictable characters will be assigned to the dummy symbol and will not appear in your documents, so you can continue to specify a list length that is greater than the actual length of the symbols.

As an alternative to using a symbol list, the DOCUMENT SET command allows you to set individual symbol values with the SYMBOL and VALUE options. The SYMBOL option specifies the name of the symbol, and the VALUE option specifies the value for that symbol. The rules for including spaces in a symbol value in a symbol list also apply to the VALUE option: you can use a simple space character or a plus sign, unless the UNESCAPED option of the DOCUMENT SET command has been specified, in which case you must use a space character. Also, the special coding that is required to include a plus sign or percent sign in symbol lists is similarly required in the VALUE option, unless the UNESCAPED option of the DOCUMENT SET command has been specified. However, ampersands, or any other character that you have specified as a symbol separator for the symbol list, have no special significance when used in the VALUE option.

Embedded template commands

The Document Handler recognises three commands that can be embedded in the template. These commands follow the syntax rules for Server Side Include commands. A Server Side Include command starts with the characters left angle bracket, exclamation mark, hyphen, hyphen, number sign followed by the command and it is terminated with the characters hyphen, hyphen, right angle bracket. For example:

```
<!--#command -->
```

The characters used to start and end the Server Side Include must be in codepage 037, otherwise the command will be ignored. The hexadecimal equivalents for these character sequences are X'4C5A60607B' and X'60606E'.

The three commands that are supported are `#set`, `#echo` and `#include`.

#set

The `#set` command is used to set the values of symbols and is useful for setting up default values for symbols. The `#set` command in the template will be ignored if the symbol has already been given a value using the DOCUMENT SET command. If a previous `#set` command has been used to assign a value to the symbol, the value will be overridden. A symbol which has been assigned a value using the DOCUMENT SET command can only be changed by issuing another DOCUMENT SET command.

#echo

The `#echo` command identifies a symbol that must be substituted when the template is inserted into the document. The string containing the `#echo` command will be completely replaced by the value associated with the symbol. If no symbol has been defined with that name, the `#echo` command will remain in the output data.

An alternative method to using the `#echo` command is to specify the symbol name, preceding it with an ampersand and terminating it with a semicolon. If we set a symbol called ASYM and give it a value of 'sample', the following two templates will give the same result after substitution.

Template 1:

```
This is an example template.  
<!--#set var=ASYM value='sample'-->  
This is a <!--#echo var=ASYM--> symbol.
```

Template 2:

```
This is an example template.  
<!--#set var=ASYM value='sample'-->  
This is a &ASYM; symbol.
```

Result of substitution:

```
This is an example template.  
This is a sample symbol.
```

#include

The `#include` command allows a template to be embedded within another template. Up to 32 levels of embedding are allowed.

For example:

```
<!--#include template=templatename-->
```

where *templatename* is the name of the template (the 48 byte name) defined in the doctemplate definition. The template name can also be enclosed in double quotes.

Using templates in your application

If you have created a template and defined it to CICS, the following example shows how you can use the template to create the contents of a document. The following template is created and defined to CICS with the name ASampleTemplate.

```
<!--#set var=ASYM value='DFLTUSER'-->
This is a sample document which has been created by user
<!--#echo var=ASYM-->.
```

In the application program, you can define a 48-byte variable called `TEMPLATENAME` and initialize it to a value of `'ASampleTemplate'`. Once again you must define a 16-byte field for the document token (in this example, `ATOKEN`). You can then issue the command to create the document.

```
EXEC CICS DOCUMENT CREATE
      DOCTOKEN(ATOKEN)
      TEMPLATE(TEMPLATENAME)
```

This will result in a document being created with the content “ This is a sample document which has been created by user DFLTUSER.”.

To change the symbol to another value, you can issue the `DOCUMENT CREATE` command with the `SYMBOLLIST` option:

```
EXEC CICS DOCUMENT CREATE
      DOCTOKEN(ATOKEN)
      TEMPLATE(TEMPLATENAME)
      SYMBOLLIST('ASYM=Joe Soap')
      LISTLENGTH(13)
```

This will result in a document being created with the content “This is a sample document which has been created by user Joe Soap.”.

The lifespan of a document

Documents created by an application exist only for the length of the CICS task in which they are created. This means that when the last program in the CICS task returns control to CICS, all documents created during the task’s lifetime are deleted. It is the application’s responsibility to save a document before terminating if the document is going to be used in another task. You can obtain a copy of the document by using the `DOCUMENT RETRIEVE`. The application can then save this copy to a location of its choice, such as a temporary storage queue. The copy can then be used to recreate the document.

The following sequence of commands show how a document can be created, retrieved and stored on a temporary storage queue, assuming that the following variables have been defined and initialised in the application program:

- A 16-byte field `ATOKEN` to hold the document token
- A 20-byte buffer `DOCBUF` to hold the retrieved document
- A fullword binary field called `FWORDLEN` to hold the length of the data retrieved
- A halfword binary field called `HWORDLEN` to hold the length for the temporary storage `WRITE` command.

```
EXEC CICS DOCUMENT CREATE
      DOCTOKEN(ATOKEN)
      TEXT('A sample document.')
      LENGTH(18)

EXEC CICS DOCUMENT RETRIEVE
      DOCTOKEN(ATOKEN)
      INTO(DOCBUF)
      LENGTH(FWORDLEN)
      MAXLENGTH(20)

EXEC CICS WRITEQ TS
      QUEUE('AQUEUE')
      FROM(DOCBUF)
      LENGTH(HWORDLEN)
```


You can now use the following sequence of commands to recreate the document in the same or another application.

```
EXEC CICS READQ TS
        QUEUE('AQUEUE')
        INTO(DOCBUF)
        LENGTH(HWORDLEN)

EXEC CICS DOCUMENT CREATE
        DOCTOKEN(ATOKEN)
        FROM(DOCBUF)
        LENGTH(FWORDLEN)
```

When the document is retrieved, the data that is delivered to the application buffer is stored in a form which contains control information necessary to reconstruct an exact replica of the document. The document that is created from the retrieved copy is therefore identical to the original document. To help the application calculate the size of the buffer needed to hold a retrieved document, each document command which alters the size of the document has a DOCSIZE option. This is a fullword value which gives the maximum size that the buffer must be to contain the document when it is retrieved. This size is calculated to include all the control information and data. The size should not be taken as an accurate size of the document as the actual length delivered to the application can often be slightly smaller than this size. The length delivered will however never exceed the length in the DOCSIZE option.

The above example introduced the use of the FROM option on the DOCUMENT CREATE command. The data passed on the FROM option was the buffer returned to the application when the DOCUMENT RETRIEVE command was issued. It is possible for the application to supply data on the FROM option that did not originate from the DOCUMENT RETRIEVE command. When this happens, the document handler treats the data as a template and parses the data for template commands and symbols.

Retrieving the document without control information

The document data containing control information is only useful to an application that wishes to recreate a copy of the original document. It is possible to issue a DOCUMENT RETRIEVE command and ask for the control information to be omitted. The following command sequence uses the DATAONLY option on the DOCUMENT RETRIEVE command to instruct the Document Handler to return only the data. This example assumes that the following variables have been defined and initialised in the application program:

- A 16-byte field ATOKEN to hold the document token
- A 20-byte buffer DOCBUF to hold the retrieved document
- A fullword binary field called FWORDLEN to hold the length of the data retrieved.

```
EXEC CICS DOCUMENT CREATE
        DOCTOKEN(ATOKEN)
        TEXT('A sample document.')
        LENGTH(18)

EXEC CICS DOCUMENT RETRIEVE
        DOCTOKEN(ATOKEN)
        INTO(DOCBUF)
        LENGTH(FWORDLEN)
        MAXLENGTH(20)
        DATAONLY
```

When the commands have executed, the buffer DOCBUF will contain the string “A sample document.”.

Constructing a document

Once a document has been created, the contents can be extended by issuing one or more DOCUMENT INSERT commands. The options on the DOCUMENT INSERT command work in the same way as the equivalent commands on the DOCUMENT CREATE command.

The following sequence of commands shows an empty document being created followed by two INSERT commands:

```
EXEC CICS DOCUMENT CREATE
      DOCTOKEN(ATOKEN)

EXEC CICS DOCUMENT INSERT
      DOCTOKEN(ATOKEN)
      TEXT('Sample line 1. ')
      LENGTH(15)

EXEC CICS DOCUMENT INSERT
      DOCTOKEN(ATOKEN)
      TEXT('Sample line 2. ')
      LENGTH(15)
```

The document resulting from the above commands will contain:

Sample line 1. Sample line 2.

You can use the DOCUMENT RETRIEVE and DOCUMENT INSERT commands to insert a whole document into an existing document. The following variables must first be defined and initialized in the application program:

- A 16-byte field RTOKEN which contains the document token of the document to be retrieved
- A buffer DOCBUF of sufficient length to hold the retrieved document
- A fullword binary field called RETRIEVLEN to hold the length of the data retrieved
- A fullword binary field called MAXLEN to hold the maximum amount of data the buffer can receive, i.e. the length of DOCBUF
- A 16-byte field ITOKEN which contains the document token of the document that is being inserted into

The following sequence of commands shows a document indicated by RTOKEN being inserted into another document indicated by ITOKEN:

```
EXEC CICS DOCUMENT RETRIEVE
      DOCTOKEN(RTOKEN)
      INTO(DOCBUF)
      LENGTH(RETRIEVLEN)
      MAXLENGTH(MAXLEN)

EXEC CICS DOCUMENT INSERT
      DOCTOKEN(ITOKEN)
      FROM(DOCBUF)
      LENGTH(RETRIEVLEN)
```

The retrieved document is inserted at the end of the document specified in the DOCUMENT INSERT command, and all the control information of the retrieved document will be present in the second document. The LENGTH parameter of the DOCUMENT INSERT command must be equal to the value returned from the DOCUMENT RETRIEVE command into the field RETRIEVLEN.

The DOCUMENT INSERT command allows an operand called SYMBOL to be used to add blocks of data to the document. SYMBOL must contain the name of a valid symbol whose value has been set. The Document Handler inserts the value that is associated with the symbol into the document.

Note that when a value associated with a symbol has been inserted into a document, you cannot change that value in the document that is being composed. If you set a different value for the symbol, the new value will be used the next time

that symbol is inserted into a document. Your change will not affect the value that has already been inserted into the document.

Using Bookmarks

The sequence in which an application inserts data into a document might not reflect the desired sequence that the data should appear in the document. Bookmarks allow the application to insert blocks of data in any order yet still control the sequence of the data in the document. A bookmark is a label that the application inserts between blocks of data. Note: a bookmark cannot be inserted in the middle of a block of data.

The following example creates a document with two blocks of text and a bookmark:

```
EXEC CICS DOCUMENT CREATE
      DOCTOKEN(ATOKEN)
      TEXT('Pre-bookmark text. ')
      LENGTH(19)

EXEC CICS DOCUMENT INSERT
      DOCTOKEN(ATOKEN)
      BOOKMARK('ABookmark      ')

EXEC CICS DOCUMENT INSERT
      DOCTOKEN(ATOKEN)
      TEXT('Post-bookmark text. ')
      LENGTH(20)
```

The document will now contain:

Pre-bookmark text. <ABookmark>Post-bookmark text.

Note that the text <ABookmark> does not appear in the document content but serves merely as a pointer to that position in the document. To add data to this document, you can insert text at the bookmark as follows:

```
EXEC CICS DOCUMENT INSERT
      DOCTOKEN(ATOKEN)
      TEXT('Inserted at a bookmark. ')
      LENGTH(25)
      AT('ABookmark      ')
```

Logically, the data of the document will contain the following (Note that in this instance, only the data is being shown and not the position of the bookmark).

Pre-bookmark text. Inserted at a bookmark. Post-bookmark text.

If the AT option is omitted, the data is always appended to the end of the document. A special bookmark of 'TOP' can be used to insert data at the top of the document, making it unnecessary to define a bookmark which will mark the top of the document.

Replacing data in the document

The following example shows how data between two bookmarks can be replaced:

```
EXEC CICS DOCUMENT CREATE
      DOCTOKEN(ATOKEN)

EXEC CICS DOCUMENT INSERT
      DOCTOKEN(ATOKEN)
      TEXT('Initial sample text. ')
      LENGTH(21)

EXEC CICS DOCUMENT INSERT
      DOCTOKEN(ATOKEN)
      BOOKMARK('BMark1          ')

EXEC CICS DOCUMENT INSERT
      DOCTOKEN(ATOKEN)
      TEXT('Text to be replaced. ')
      LENGTH(21)

EXEC CICS DOCUMENT INSERT
      DOCTOKEN(ATOKEN)
      BOOKMARK('BMark2          ')

EXEC CICS DOCUMENT INSERT
      DOCTOKEN(ATOKEN)
      TEXT('Final sample text. ')
      LENGTH(19)
```

At this point the logical structure of the document will be as follows:

```
Initial sample text. <BMark1>Text to be replaced. <BMark2>Final
sample text.
```

You can now issue the command to replace the text between the two bookmarks, BMark1 and BMark2:

```
EXEC CICS DOCUMENT INSERT
      DOCTOKEN(ATOKEN)
      TEXT('Replacement Text. ')
      LENGTH(18)
      AT('BMark1          ')
      TO('BMark2          ')
```

The document now has the following logical structure:

```
Initial sample text. <BMark1>Replacement Text. <BMark2>Final
sample text.
```

Code page conversion for documents

The documents that an application creates may be transmitted to systems running on other platforms; for example, when a document is used to supply a Web page to a client. Textual data that is in the code pages used by the CICS system must be converted to the code pages used on the target system. This process is known as code page conversion. A code page used by the CICS system is usually described as a host code page, except when CICS is acting as an HTTP client. A code page used by the target system is described as a client code page, or where the target system is a Web client or server using ASCII, it can be referred to as a character set.

You can make CICS documents include information about the code pages in which they have been created. When you create a document using the EXEC CICS DOCUMENT CREATE and EXEC CICS DOCUMENT INSERT commands, you can use the HOSTCODEPAGE option together with any of the TEXT, FROM, TEMPLATE and SYMBOL options, to indicate the code page for that block of data. Each block can be specified in a different code page.

When you use the EXEC CICS DOCUMENT RETRIEVE command to retrieve a document for sending, you can specify the CLNTCODEPAGE option to make the Document Handler to convert all the individual blocks from their respective host code pages, into a single client code page that is suitable for use by the target system.

| For CICS Web support, when a CICS document is specified for sending by a
| Web-aware application program using the EXEC CICS WEB API commands, the
| EXEC CICS DOCUMENT RETRIEVE command is not used by the application
| program. Instead, the document token of the CICS document is specified, and CICS
| manages retrieval of the document. Conversion to a client code page is handled by
| CICS, according to options that the application program specifies on the EXEC
| CICS WEB API commands.

| Also for CICS Web support, when a CICS document template is specified in a
| URIMAP definition to provide a static response to a Web client, conversion to a
| client code page is handled by CICS. The host code page in which the template
| exists, and the client code page to which it should be converted, are specified in the
| URIMAP definition. When the static response is required, CICS creates a document
| using the template, retrieves the document, and carries out appropriate code page
| conversion.

| CICS documents and document templates cannot be converted to or from the
| UTF-8 and UTF-16 character encodings. This restriction applies whether they are
| used as a static response in CICS Web support, retrieved by CICS in response to
| EXEC CICS WEB API commands, or retrieved by an application program using an
| EXEC CICS DOCUMENT RETRIEVE command.

Chapter 25. Named counter servers

This chapter describes the services provided by CICS named counter servers.

CICS provides a facility for generating unique sequence numbers for use by application programs in a Parallel Sysplex environment. This facility is controlled by a named counter server, which maintains each sequence of numbers as a **named counter**. Each time a sequence number is assigned, the corresponding named counter is incremented automatically. By default, the increment is 1, ensuring that the next request gets the next number in sequence. You can vary the increment when using the EXEC CICS GET command to request the next number.

There are various uses for this facility, such as obtaining a unique number for documents (for example, customer orders, invoices, and despatch notes), or for obtaining a block of numbers for allocating customer record numbers in a customer file.

In a single CICS region, there are various methods you can use to control the allocation of a unique number. For example, you could use the CICS common work area (CWA) to store a number that is updated by each application program that uses the number. The problem with the CWA method is that the CWA is unique to the CICS address space, and cannot be shared by other regions that are running the same application. A CICS shared data table could be used to provide such a service, but the CICS regions would all have to reside in the same MVS image. The named counter facility overcomes all the sharing difficulties presented by other methods by maintaining its named counters in the coupling facility, and providing access through a named counter server running in each MVS image in the sysplex. This ensures that all CICS regions throughout the Parallel Sysplex have access to the same named counters.

When you use a named counter server, each normal request (to assign the next counter value) only requires a single coupling facility access. This provides a significant improvement in performance compared to the use of files for this purpose. The named counter server also performs better than coupling facility data tables in this respect, because at least two coupling facility accesses are required to update a coupling facility data table. Depending on your hardware configuration, you should easily be able to make many thousands of named counter server requests each second.

This chapter describes:

- “The named counter fields”
- “Named counter pools” on page 342
- “Using the named counter EXEC interface” on page 343
- “Using the named counter CALL interface” on page 345
- “Named counter recovery” on page 358

The named counter fields

Each named counter consists of:

The counter name

The name can be up to 16-bytes, comprising the characters A through Z, 0 through 9, \$ @ # and _. Names less than 16 bytes should be padded with trailing blanks.

The current value

The next number to be assigned to a requesting application program.

The minimum value

Specifies the minimum number for a counter, and the number to which a counter is reset by the server in response to a REWIND command.

The maximum value

Specifies the maximum number that can be assigned by a counter, after which the counter must be explicitly reset by a REWIND command (or automatically by the WRAP option).

All values are stored internally as 8-byte (doubleword) binary numbers. The EXEC CICS interface allows you to use them as either fullword signed binary numbers or doubleword unsigned binary numbers. This can give rise to overflow conditions if you define a named counter using the doubleword command (see “Using the named counter EXEC interface” on page 343) and request numbers from the server using the signed fullword version of the command.

Named counter pools

A named counter is stored in a named counter pool, which resides in a list structure in a coupling facility. Each pool, even if its list structure is defined with the minimum size of 256KB, can hold up to a thousand named counters.

You create a named counter pool by defining the coupling facility list structure for the pool, and then starting the first named counter server for the pool. Pool names are of 1 to 8 bytes from the same character set for counter names. Although pool names can be made up from any of the allowed characters, names of the form DFHNCxxx are recommended.

You can create different pools to suit your needs. You could create a pool for use by production CICS regions (for example, called DFHNCPRD), and others for test and development regions (for example, using names like DFHNCTST and DFHNCDEV). See “Named counter options table” for information about how you can use logical pool names in your application programs, and how these are resolved to actual pool names at runtime.

Defining a list structure for a named counter server, and starting a named counter server, is explained in the *CICS System Definition Guide*.

Named counter options table

The POOL(*name*) parameter is optional on all the EXEC CICS COUNTER and DCOUNTER commands (see “Using the named counter EXEC interface” on page 343 for more information). If you specify the POOL parameter, it can refer to either an actual or a logical pool name. Whether you specify a POOL parameter or omit it, CICS resolves the actual pool name by reference to the named counter options table, which is loaded from the link list.

The named counter options table, DFHNCOPT, provides several methods for determining the actual pool name referenced by a named counter API command, all of which are described in the *CICS System Definition Guide*. This also describes the DFHNCO macro that you can use to create your own options table.

This section discusses how the POOLSEL parameter in the default options table works in conjunction with the POOL(*name*) option on the API. The default options table is supplied in source and object form. The pregenerated version is in *hlq.SDFHLINK*, and the source version, which is supplied in the *hlq.SDFHSAMP* library (where *hlq* represents the high-level qualifier for the library names, established at CICS installation time), contains the following entries:

```
DFHNCO    POOLSEL=DFHNC*,POOL=YES
DFHNCO    POOL=
END        DFHNCOPT
```

The default options table entries work as follows:

POOLSEL=DFHNC*

This pool selection parameter defines a generic logical pool name beginning with the letters DFHNC. If any named counter API request specifies a pool name that matches this generic name, the pool name is determined by the POOL= operand in the DFHNCO entry. Because this is POOL=YES in the default table, the name passed on the POOL(*name*) option of the API command is taken to be an actual name. Thus, the default options table specifies that all logical pool names beginning with DFHNC are actual pool names.

POOL=

This entry in the default table is the 'default' entry. Because the POOLSEL parameter is not specified, it defaults to POOLSEL=*, which means it is taken to match any value on a POOL parameter that does not find a more explicit match. Thus, any named counter API request that:

- Specifies a POOL value that begins with other than DFHNC, or
- Omits the POOL name parameter altogether

is mapped to the the default pool (indicated by a POOL= options table parameter that omits a name operand).

You can specify the default pool name to be used by a CICS region by specifying the NCPLDFT system initialization parameter. If NCPLDFT is omitted, the pool name defaults to DFHNC001.

You can see from the above that you do not need to create you own options table, and named counter API commands do not need to specify the POOL option, if:

- You use pool names of the form DFHNCxxx, or
- Your CICS region uses only one pool that can be defined by the NCPLDFT system initialization parameter.

Notes:

1. DFHNCOPT named counter options tables are not suffixed. A CICS region loads the first table found in the MVS link list.
2. There must be a named counter server running, in the same MVS image as your CICS region, for each named counter pool used by your application programs.

Using the named counter EXEC interface

Although all named counter values are held internally as doubleword unsigned binary numbers, the CICS API provides both a fullword (COUNTER) and doubleword (DCOUNTER) set of commands, which you should not mix. These EXEC CICS commands allow you to perform the following operations on named counters:

DEFINE

Defines a new named counter, setting minimum and maximum values, and specifying the current number at which the counter is to start.

DELETE

Deletes a named counter from its named counter pool.

GET

Gets the current number from the named counter, provided the maximum number has not already been allocated.

Using the WRAP option: If the maximum number has been allocated to a previous request, the counter is in a counter-at-limit condition and the request fails, unless you specify the WRAP option. This option specifies that a counter in the counter-at-limit condition is to be reset automatically to its defined minimum value. After the reset, the minimum value is returned as the current number, and the counter is updated ready for the next request.

Using the INCREMENT option: By default, a named counter is updated by an increment of 1, after the server has assigned the current number to a GET request. If you want more than one number at a time, you can specify the INCREMENT option, which effectively reserves a block of numbers from the current number. For example, if you specify INCREMENT(50), and the server returns 100 025:

- Your application program can use 100 025 through 100 074
- As a result of updating the current number (100 025) by 50, the current number is left at 100 075 ready for the next request.

This example assumes that updating the current value by the INCREMENT(50) option does not exceed the maximum value by more than 1. If the range of numbers between the current value and the maximum value plus 1 is less than the specified increment, the request fails unless you also specify the REDUCE option.

Using the REDUCE option: To ensure that a request does not fail because the remaining range of numbers is too small to satisfy your INCREMENT value (the current number is too near the maximum value), specify the REDUCE option. With the reduce option, the server automatically adjusts the increment to allow it to assign all the remaining numbers, leaving the counter in the counter-at-limit condition.

Using both the WRAP and REDUCE options: If you specify both options, only one is effective depending on the state of the counter:

- If the counter is already at its limit when the server receives the GET request, the REDUCE option has no effect and the WRAP option is obeyed.
- If the counter is not quite at its limit when the server receives the GET request, but the remaining range is too small for increment, the REDUCE option is obeyed and the WRAP option has no effect.

Using the COMPAREMIN and COMPAREMAX options: You can use these options to make the named counter GET (and UPDATE) operation conditional upon the current number being within a specified range, or being greater than, or less than, one of the specified comparison values.

QUERY

Queries the named counter to obtain the current, minimum, and maximum values. Note that you cannot use more than one named counter command in a way that is atomic, and you cannot rely on the information returned on a QUERY command not having been changed by another task somewhere in the

sysplex. Even the CICS sysplex-wide ENQ facility cannot lock a counter for you, because a named counter could be accessed by a batch application program using the named counter CALL interface. If you want to make an operation conditional upon the current value being within a certain range, or greater than, or less than, a certain number, use the COMPAREMIN and COMPAREMAX parameters on your request.

REWIND

Rewinds a named counter that is in the counter-at-limit condition back to its defined minimum value.

UPDATE

Updates the current value of a named counter to a new current value. For example, you could set the current value to the next free key in a database. Like the GET command, this can be made conditional by specifying COMPAREMIN and COMPAREMAX values.

Using the named counter CALL interface

In addition to the CICS named counter API, CICS provides a call interface that you can use from a batch application to access the same named counters. This could be important where you have an application that uses both CICS and batch programs, and both need to access the same named counter to obtain unique numbers from a specified range. The call interface does not depend on CICS services, therefore it can also be used in applications running under any release of CICS.

The named counter call interface does not use CICS command-level API, therefore the system initialization parameter CMDPROT=YES has no effect. If the interface is called from a CICS application program that is executing in user key, it switches to CICS key while processing the request but CICS does not attempt to verify that the program has write access to the output parameter fields.

The first request by an application region that addresses a particular pool automatically establishes a connection to the server for that pool. This connection is associated with the current MVS TCB (which for CICS is the quasi-reentrant (QR) TCB) and normally lasts until the TCB terminates at end of job. This connection can only be used from the TCB under which the connection was established. A request issued from another TCB will establish a separate connection to the server.

Note: The named counter server interface uses MVS name/token services internally. A consequence of this is that jobs using the named counter interface cannot use MVS checkpoint/restart services (as described in APAR OW06685).

Application programming considerations

To use the named counter callable interface:

1. Ensure your application programs include the appropriate copybook that defines the parameter list definition for the application programming language. The copybook defines symbolic constants for the function codes and return codes, and also defines the callable entry point for high level languages. The copybook name is of the form DFHNCxxx where xxx indicates the programming language, as follows:

ASM or EQU for Assembler
 C for C/C++
 COB for COBOL
 PLI for PL/I

2. Ensure the application program is link-edited with the callable interface linkage routine, DFHNCTR.
3. Ensure the named counter server interface module, DFHNCIF, and the options table, DFHNCOPT, are available to the CICS region. That is, these objects must be in a STEPLIB library, in a linklist library, or in the LPA. To support CICS application programs that run in user key, DFHNCIF must be loaded from an APF-authorized library. The default option table and the named counter server interface module are supplied in CICSTS31.CICS.SDFHLINK.

CICS provides copybooks for all the supported languages:

Assembler

The standard assembler named counter interface definitions are provided in copybook DFHNCAASM. Include these in your application programs using COPY DFHNCAASM within a constant CSECT area. The symbolic values are defined as static fullword constants, in the form NC_name DC F'nnn'. For example:

```
NC_BROWSE_NEXT DC F'7'
```

An alternative set of definitions is provided as symbolic equated values in copy book DFHNCEQU. These symbols are all of the form NC_EQU_name to avoid conflict with the static constants. Note that when these equated values are used for function codes or return code comparisons, they should be used as address constant values, so that for example the function code NC_ASSIGN can be replaced by a reference to =A(NC_EQU_ASSIGN).

The syntax of the assembler version of the call to the named counter interface is as follows:

```
CALL DFHNCTR,(function,return_code,pool_selector,counter_name,      X
                value_length,current_value,minimum_value,maximum_value, X
                counter_options,update_value,compare_min,compare_max),VL
```

The CALL macro must specify the VL option to set the end of list indication, as shown in the following example:

```
CALL DFHNCTR,(NC_ASSIGN,RC,POOL,NAME,CTRLLEN,CTR),VL
```

C/C++

The named counter interface definitions for C/C++ are provided in header file DFHNCC. The symbolic constant names are in upper case. The function name is dfhnctr, in lower case.

COBOL

The named counter interface definitions for COBOL are provided in copybook DFHNCCOB.

COBOL does not allow underscores within names, therefore the symbolic names provided in copy book DFHNCCOB use a hyphen instead of an underscore (for example NC-ASSIGN and NC-COUNTER-AT-LIMIT).

Note that the RETURN-CODE special register is set by each call, which affects the program overall return code if it is not explicitly set again before the program terminates.

PL/I

The named counter interface definitions for PL/I are provided in include file DFHNCPLI.

Syntax

The syntax of the named counter call is as follows:

```
CALL DFHNCTR(function,return_code,pool_selector,counter_name,  
             value_length,current_value,minimum_value,maximum_value,  
             counter_options,update_value,compare_min,compare_max);
```

Figure 82. DFHNCTR call syntax-PL/I illustration

Notes:

1. All functions that refer to a named counter require at least the first four parameters, but the remaining parameters are optional, and trailing unused parameters can be omitted.

If you do not want to use an imbedded optional parameter, either specify the default value or ensure that the parameter list contains a null address for the omitted parameter. For an example of a call that omits an optional parameter, see “Example of DFHNCTR calls with null parameters” on page 353.

2. The NC_FINISH function requires the first three parameters only.

function

specifies the function to be performed, as a 32-bit integer, using one of the following symbolic constants.

NC_CREATE Create a new named counter, using the initial value, range limits, and default options specified on the *current_value*, *minimum_value*, *maximum_value*, *update_value* and *counter_options* parameters.

If you omit an optional value parameter, the new named counter is created using the default for the omitted value. For example, if you omit all the optional parameters, the counter is created with an initial value of 0, a minimum value of 0, and a maximum value of high values (the double word field is filled with X'FF').

NC_ASSIGN Assign the current value of the named counter, then increment it ready for the next request. When the number assigned equals the maximum number specified for the counter, it is incremented finally to a value 1 greater than the maximum. This ensures that any subsequent NC_ASSIGN requests for the named counter fail (with NC_COUNTER_AT_LIMIT) until the counter is reset using the NC_REWIND function, or automatically rewound by the NC_WRAP counter option (see the *counter_options* parameter).

This operation can include a conditional test on the current value of the named counter, using the *compare_min* and *compare_max* parameters.

The server returns the minimum and maximum values if you specify these fields on the call parameter list and the request is successful.

You can use the *counter_options* parameter on the NC_ASSIGN request to override the counter options set by the NC_CREATE request.

You can use the *update_value* parameter to specify the increment to be used on this request only for the named counter (the default increment is 1). This enables you to obtain

a range of numbers on a single request. For more information, see the description of the *update_value* parameter.

Note that the named counter is incremented by *update_value* after the current value is assigned. For example:

If the current value is 109
and *update_value* specifies 25

the named counter server returns 109 and sets the current value to 134 ready for the next NC_ASSIGN request, effectively assigning numbers in the range 109 through 133. The increment can be any value between zero and the limit is determined by the minimum and maximum values set for the named counter. Thus the increment limit is $((\text{maximum_value} + 1) - \text{minimum_value})$. An increment of zero causes NC_ASSIGN to operate the same as NC_INQUIRE, except for any comparison options.

When the increment is greater than 1, and the named counter is near the maximum limit, the server may not be able to increment the current number by the whole value of the specified increment. This situation occurs when incrementing the counter would exceed the maximum value plus 1. You control what action the named counter server takes in this situation through the *counter_options* NC_NOREDUCE | NC_REDUCE, and NC_NOWRAP | NC_WRAP. See *counter_options* parameter for information about the operation of these options.

NC_BROWSE_FIRST

Return the details of the first named counter with a name greater than or equal to the specified name, and update the *counter_name* field accordingly.

NC_BROWSE_NEXT

Return the details of the next named counter with a name greater than the specified name, and update the *counter_name* field accordingly.

NC_DELETE

Delete the specified named counter. The server returns the *current_value*, *minimum_value*, *maximum_value*, and *counter_options* if you specify these fields on the parameter list and the request is successful.

NC_FINISH

Terminate the connection between the current MVS task (TCB) and the named counter server for the specified pool. If a further request is made to the same pool, a new connection is established.

This function does not apply to a specific counter, therefore the only parameters required are the function, the return code and the pool name.

Use this function only when there is a special reason to terminate the connection (for example, to allow the server to be shut down).

NC_INQUIRE

Return the details (*current_value*, *minimum_value*, *maximum_value* and *counter_options*) of a named counter

without modifying it. The current value is the value to be returned on the next NC_ASSIGN call. If the maximum value of the named counter has already been assigned, the server returns a current value one greater than the maximum value.

NC_REWIND Reset the named counter to its minimum value. This function is valid only when the last number permitted by the maximum value has been assigned, leaving the counter in the NC_COUNTER_AT_LIMIT condition. If an NC_ASSIGN call causes the server to assign the last number for the named counter, use the NC_REWIND function to reset the counter.

This operation can include a conditional test on the current value of the named counter, using the *compare_min* and *compare_max* parameters.

The server returns the new current value, minimum value, and maximum value if you specify these fields on the parameter list and the request is successful.

If any option parameter or *update_value* parameter was specified on an NC_ASSIGN request which failed because the named counter was at its limit, the same parameter values must also be specified on the NC_REWIND request, so that it can check whether the original NC_ASSIGN would still fail. The NC_REWIND request is suppressed with return code 102 (NC_COUNTER_NOT_AT_LIMIT) whenever the corresponding NC_ASSIGN request would succeed.

If the NC_WRAP option is in effect, or the *update_value* parameter is zero, NC_REWIND is suppressed because NC_ASSIGN always succeeds with these conditions. See the *counter_options* parameter for information about the NC_WRAP option.

NC_UPDATE Set the named counter to a new value. This operation can include a conditional test on the current value of the named counter, using the *compare_min* and *compare_max* parameters.

You specify the new value on the *update_value* parameter. If you don't specify a new value, the named counter remains unchanged.

You can specify a valid *counter_options* override parameter (or a null address) with this function, but counter options have no effect. Specify either a null address or NC_NONE as the *counter_options* parameter.

return_code

specifies a 32-bit integer field to receive the return code. The same information is also returned in register 15, which for COBOL callers is stored in the RETURN-CODE special register.

Each return code has a corresponding symbolic constant. See "Return codes" on page 355 for details of these.

pool_selector

specifies an 8-character pool selection parameter that you use to identify the pool in which the named counter resides.

This parameter is optional. If you omit the pool selection parameter, a string of 8 blank X'40') characters is assumed.

The acceptable characters for *pool_selector* are A through Z, 0 through 9, \$ @ # and _ (underscore) but the first character cannot be numeric or an underscore. The parameter should be padded to fill 8 characters with trailing spaces as necessary. The parameter can be all spaces to use the default pool for the current region, provided this is mapped by the options table to a valid non-blank named counter name).

Depending on the named counter options table in use, you can use the pool selector parameter either as an actual pool name, or as a logical pool name that is mapped to a real pool name through the options table. The default options table assumes:

- That any pool selection parameter beginning with DFHNC (matching the table entry with POOLSEL=DFHNC*) is an actual pool name
- That any other pool selection parameter (including all blanks) maps to the default pool name.

Note: The default pool name for the call interface is DFHNC001. The default pool name for the EXEC CICS API is defined by the NCPLDFT system initialization parameter.

See “Named counter options table” on page 342 for information about the pool selection parameter in the DFHNCOPT options table.

counter_name

specifies a 16-byte field containing the name of the named counter, padded if necessary with trailing spaces.

The acceptable characters for the name are A through Z, 0 through 9, \$ @ # and _ (underscore) but the first character cannot be numeric or an underscore. The parameter should be padded to fill 16 characters with trailing spaces as necessary.

You are recommended to use names that have a common prefix of up to 8 bytes to avoid conflicts with other applications. Any named counters used internally by CICS have names starting with DFH.

For the NC_BROWSE_FIRST and NC_BROWSE_NEXT functions, the actual name is returned in this field, which must be a variable for this purpose. For all other functions, this can be a constant.

value_length

specifies a 32-bit integer field indicating the length of each named counter value field. Values can be in unsigned format (where the high order bit is part of the value), or in positive signed format (where the high order bit is reserved for a zero sign bit). To use unsigned format, specify the length in bytes, in the range 1 to 8, corresponding to 8 to 64 bits. To use values in signed format, specify the length in bytes as a negative number in the range –1 to –8, corresponding to 7 to 63 bits. For compatibility with the named counter EXEC interface, this should be set to –4 for counters that are handled as fullword signed binary values (COUNTER) and 8 for counters that are handled as doubleword unsigned binary values (DCOUNTER).

If no value parameters are used on the call, you can either omit all the trailing unused parameters completely, including the value length, or specify *value_length* as 0.

When input values are shorter than 8 bytes, they are extended with high-order zero bytes to the full 8 bytes used internally. When output values are returned in a short value field, the specified number of low-order bytes are returned, ignoring any higher-order bytes. However, if the value is too large to be

represented correctly within the field, then a warning return code might be set, as described in “Checking for result overflow” on page 353.

current_value

specifies a variable to be used for:

- Setting the initial sequence number for the named counter
- Receiving the current sequence number from the named counter.

For the NC_CREATE function this parameter is an input (sender) field and can be defined as a constant. The default value is low values (binary zeroes). The value can either be within the range specified by the following minimum and maximum values, or it can be one greater than the maximum value, in which case the counter has to be reset using the NC_REWIND function before it is used. No sign check is made for this field, but a value that has the sign bit set would normally be rejected by the server as being inconsistent with the counter limits. For a counter that has a range consisting of all signed numbers, the counter at limit value does have the sign bit set, and this can be used as a valid input value.

For all other counter functions, this parameter is an output (receiver) field and must be defined as a variable.

minimum_value

specifies a variable to be used for:

- Setting the minimum value for the named counter
- Receiving from the named counter the specified minimum value.

For the NC_CREATE function this parameter is an input (sender) field and can be defined as a constant. The default value is low values (binary zeroes).

For all other functions, this parameter is an output (receiver) field and must be defined as a variable.

maximum_value

specifies a variable to be used for:

- Setting the maximum value for the named counter
- Receiving from the named counter the specified maximum value.

For the NC_CREATE function this parameter is an input (sender) field and can be defined as a constant. If you specify a non-zero *value_length* parameter but omit *maximum_value*, then *maximum_value* defaults to high values (or, for signed variables, the largest positive value) for the specified length. If the *value_length* parameter is omitted or is specified as zero, then *maximum_value* defaults to eight bytes of high values. However, if the minimum value is all low values and the maximum value is eight bytes of high values, the maximum value is reduced to allow some reserved values to be available to the server for internal use.

For all other functions, this parameter is an output (receiver) field and must be defined as a variable.

counter_options

specifies an optional fullword field to indicate named counter options that control wrapping and increment reducing. The valid options are represented by the symbolic values NC_WRAP or NC_NOWRAP and NC_REDUCE or NC_NOREDUCE. The default options are NC_NOWRAP and NC_NOREDUCE.

NC_NOWRAP The server does not automatically rewind the named counter back to the minimum value in response to an NC_ASSIGN

request that fails with the NC_COUNTER_AT_LIMIT condition. With NC_NOWRAP in force, and the named counter in the NC_COUNTER_AT_LIMIT condition, the NC_ASSIGN function is inoperative until the counter is reset by an NC_REWIND request (or the counter option reset to NC_WRAP).

NC_WRAP The server automatically performs an NC_REWIND in response to an NC_ASSIGN request for a counter that is in the NC_COUNTER_AT_LIMIT condition. The server sets the current value of the named counter equal to the minimum value, returns the new current value to the caller, and increments the named counter.

NC_NOREDUCE If the range of numbers remaining to be assigned (the difference between the current value and the maximum value plus 1) is less than the increment specified on the *update_value* parameter, the assign fails (unless NC_WRAP is in force). NC_NOREDUCE, with NC_NOWRAP, means the NC_ASSIGN request fails with the NC_COUNTER_AT_LIMIT condition.

For example, if a request specifies an update value of 15 when the current number is 199 990 and the counter maximum number is defined as 199 999, the NC_REQUEST fails because the increment would cause the current number to exceed 200 000.

NC_REDUCE If the range of numbers remaining to be assigned (the difference between the current value and the maximum value plus 1) is less than the increment specified on the *update_value* parameter, the increment is reduced and the assign succeeds. In this case, the NC_ASSIGN request has been assigned a range of numbers less than that specified by the *update_value*, and the named counter is left in the NC_COUNTER_AT_LIMIT condition. Subsequent NC_ASSIGN requests will fail until the named counter is reset with an NC_REWIND request.

The options specified on NC_CREATE are stored with the named counter and are used as the defaults for other named counter functions. You can override the options on NC_ASSIGN, NC_REWIND or NC_UPDATE requests. If you don't want to specify *counter_options* on a DFHNCTR call, specify the symbolic constant NC_NONE (equal to zero) as the input parameter (or specify a null address).

For the NC_CREATE, NC_ASSIGN, NC_REWIND, and NC_UPDATE functions, this parameter is an input field.

For the NC_DELETE, NC_INQUIRE, and NC_BROWSE functions, this parameter is an output field, which returns the options specified on NC_CREATE.

update_value

specifies the value to be used to update the counter. For NC_ASSIGN, this is the increment to be added to the current counter value (after the current number is assigned). See the NC_ASSIGN option on the *function* parameter for information on how specifying an increment other than 1 can affect an assign operation.

For NC_UPDATE, this is the new current value for the named counter.

compare_min

specifies a value to be compared with the named counter's current value. If you specify a value, this parameter makes the NC_ASSIGN, NC_REWIND or NC_UPDATE operation conditional on the current value of the named counter being greater than or equal to the specified value. If the comparison is not satisfied, the operation is rejected with a counter-out-of-range return code (RC 103).

If you omit this parameter by specifying a null address, the server does not perform the comparison.

compare_max

specifies a value to be compared with the named counter's current value. If you specify a value, this parameter makes the NC_ASSIGN, NC_REWIND or NC_UPDATE operation conditional on the current value of the named counter being less than or equal to the specified value. If the comparison is not satisfied, the operation is rejected with a counter-out-of-range return code (RC 103).

If you specify high values (X'FF') for this parameter, the server does not perform the comparison. You must specify (X'FF') in all the bytes specified by the *value_length* parameter.

If the *compare_max* value is less than the *compare_min* value, the valid range is assumed to wrap round, in which case the current value is considered to be in range if it satisfies either comparison, otherwise both comparisons must be satisfied.

Checking for result overflow

The call interface checks for results which do not fit into the specified size of result field, or which overflow into the sign bit when signed variables are used.

If a result field (*counter_value*, *minimum_value* or *maximum_value*) has been defined as a signed variable by specifying *value_length* as a negative value, the call interface checks for results that overflow into the sign bit. In this case, the operation completes normally but the return code NC_RESULT_OVERFLOW is set. As a special case, a result value for a counter which is at its limit value is not checked for this form of overflow, to avoid setting the return code unnecessarily. This means that if a query is made to a counter which is at its limit, and whose maximum value is the maximum positive value, a negative number might be returned as the current counter value without causing this return code.

If a result field (*counter_value*, *minimum_value* or *maximum_value*) is too short to contain the full non-zero part of the result, the operation completes normally, but one of the following return codes is set:

- NC_RESULT_CARRY, if the leading part is exactly equal to 1.
- NC_RESULT_TRUNCATED, if the leading part is greater than 1.

If a 4-byte unsigned counter that has a maximum of high values has reached its limit value, the return code NC_RESULT_CARRY is set, and the counter value is zero.

Example of DFHNCTR calls with null parameters

If you omit an optional parameter on a DFHNCTR call, ensure that the parameter list is built with a null address for the missing parameter. The example that follows illustrates how to issue, from a COBOL program, an NC_CREATE request with some parameters set to null addresses.

DFHNCTR call with null addresses for omitted parameters: In this example, the parameters used on the call are defined in the WORKING-STORAGE SECTION, as follows:

Call parameter	COBOL variable	Field definition
<i>function</i>	01 FUNCTION	PIC S9(8) COMP VALUE +1.
<i>return_code</i>	01 NC-RETURN-CODE.	PIC S9(8) COMP VALUE +0.
<i>pool_selector</i>	01 NC-POOL-SELECTOR	PIC X(8).
<i>counter_name</i>	01 NC-COUNTER-NAME	PIC X(16).
<i>value_length</i>	01 NC_VALUE-LENGTH	PIC S9(8) COMP VALUE +4.
<i>current_value</i>	01 NC-CURRENT-VALUE	PIC S9(8) VALUE +0.
<i>minimum_value</i>	01 NC-MIN-VALUE	PIC S9(8) VALUE +0.
<i>maximum_value</i>	01 NC-MAX-VALUE	PIC S9(8) VALUE -1.
<i>counter_options</i>	01 NC-OPTIONS	PIC S9(8) COMP VALUE +0.
<i>update_value</i>	01 NC-UPDATE-VALUE	PIC S9(8) VALUE +1.
<i>compare_min</i>	01 NC-COMP-MIN	PIC S9(8) VALUE +0.
<i>compare_max</i>	01 NC-COMP-MAX	PIC S9(8) VALUE +0.

The variable used for the null address is defined in the LINKAGE SECTION, as follows:

```

LINKAGE SECTION.
01  NULL-PTR          USAGE IS POINTER.

```

Using the data names specified in the WORKING-STORAGE SECTION as described above, and the NULL-PTR name as described in the LINKAGE SECTION, the following illustrates a call to a named counter server where *value_length*, *current_value*, *minimum_value* and *counter_options* are the only optional parameters specified. The others are allowed to default, or, in the case of trailing optional parameters, omitted altogether.

```

NAMED-COUNTER SECTION.
*
    SET ADDRESS OF NULL-PTR TO NULLS.
*
    MOVE 1      TO FUNCTION.
    MOVE 100    TO NC-MIN-VALUE NC-CURRENT-VALUE.
    MOVE NC-WRAP TO NC-OPTIONS.
    MOVE "DFHNC001" TO NC-POOL-SELECTOR.
    MOVE "CUSTOMER_NUMBER" TO NC-COUNTER-NAME.
    CALL 'DFHNCTR' USING FUNCTION NC-RETURN-CODE NC-POOL-SELECTOR
                        NC-COUNTER-NAME NC-VALUE-LENGTH NC-CURRENT-VALUE
                        NC-MIN-VALUE NULL-PTR NC-OPTIONS.

```

Return codes

There are three warning return codes (1–3) for the named counter call interface, which indicate result overflows for a request that otherwise completed normally. If more than one warning return code applies for the same request, the highest applicable warning return code is set.

The remaining return codes are divided into ranges (100, 200, 300, and 400®) according to their severity. Each range of non-zero return codes begins with a dummy return code that describes the return code category, to make it easy to check for values in each range using a symbolic name.

In the list that follows, the numeric return code is followed by its symbolic name.

0 (NC_OK)

The request completed normally.

1 (NC_RESULT_OVERFLOW)

The result value overflowed into the sign bit.

2 (NC_RESULT_CARRY)

The result value overflowed, and the leading part was exactly equal to 1.

3 (NC_RESULT_TRUNCATED)

The result value overflowed, and the leading part was greater than 1.

100 (NC_COND)

Return codes in this range indicate that a conditional function did not succeed because the condition was not satisfied:

101 (NC_COUNTER_AT_LIMIT)

An NC_ASSIGN function is rejected because the previous request for this named counter obtained the maximum value and the counter is now at its limit. New counter values cannot be assigned until an NC_REWIND function call is issued to reset the counter.

102 (NC_COUNTER_NOT_AT_LIMIT)

An NC_REWIND FUNCTION is rejected because the named counter is not at its limit value. This is most likely to occur when another task has already succeeded in resetting the counter with an NC_REWIND.

103 (NC_COUNTER_OUT_OF_RANGE)

The current value of the named counter is not within the range specified on the *compare_min* and *compare_max* parameters.

200 (NC_EXCEPTION)

Return codes in this range indicate an exception condition that an application program should be able to handle:

201 (NC_COUNTER_NOT_FOUND)

The named counter cannot be found.

202 (NC_DUPLICATE_COUNTER_NAME)

An NC_CREATE function is rejected because a named counter with the specified name already exists.

203 (NC_SERVER_NOT_CONNECTED)

An NC_FINISH function is rejected because no active connection exists for the selected pool.

300 (NC_ENVIRONMENT_ERROR)

Return codes in this range indicate an environment error. These are serious errors, normally caused by some external factor, which a program may not be able to handle.

301 (NC_UNKNOWN_ERROR)

The server has reported an error code that is not understood by the interface. Generally, this is not possible unless the interface load module, DFHNCIF, is at a lower maintenance or release level than the server itself.

302 (NC_NO_SPACE_IN_POOL)

A new counter cannot be created because there is insufficient space in the named counter pool.

303 (NC_CF_ACCESS_ERROR)

An unexpected error, such as structure failure or loss of connectivity, has occurred on a macro used to access the coupling facility. Further information can be found in message DFHNC0441 in the CICS job log.

304 (NC_NO_SERVER_SELECTED)

The pool selection parameter specified in the program cannot be resolved to a valid server name using the current options table.

305 (NC_SERVER_NOT_AVAILABLE)

The interface is unable to establish a connection to the server for the appropriate named counter pool. Further information can be found in an AXM services message in the CICS job log.

306 (NC_SERVER_REQUEST_FAILED)

An abend occurred during server processing of a request. Further information can be found in a message in the CICS job log and the server job log.

307 (NC_NAME_TOKEN_ERROR)

An IEANTxx name/token service call within the named counter interface module gave an unexpected return code.

308 (NC_OPTION_TABLE_NOT_FOUND)

The DFHNCOPT options table module, required for resolving a pool name, could not be loaded.

309 (NC_OPTION_TABLE_INVALID)

During processing of the options table, the named counter interface encountered an unknown entry format. Either the options table is not correctly generated, or the DFHNCIF interface load module is not at the same release level as the options table.

310 (NC_USER_EXIT_NOT_FOUND)

An options table entry matching the given pool name specified a user exit program, but the user exit program is not link-edited with the options table and cannot be loaded.

311 (NC_STRUCTURE_UNAVAILABLE)

The named counter server list structure is temporarily unavailable. One reason, for example, for this is that a z/OS system-managed rebuild is in progress.

Note: The EXEC CICS interface to the named counter uses the CALL interface internally, but it hides this return code from the application program by waiting for one second and retrying the request. The EXEC CICS interface continues this wait and retry until it succeeds, with the result that the application program sees only a time delay, not an error response. You can use the same technique in your application programs that use the CALL interface.

400 (NC_PARAMETER_ERROR)

Return codes in this range indicate a parameter error, generally the result of a coding error in the calling program.

401 (NC_INVALID_PARAMETER_LIST)

The parameter list is invalid for one of the following reasons:

- Too few parameters are specified (less than four, or less than three for the NC_FINISH function)
- Too many parameters are given (more than eight)
- A parameter address is zero
- The end-of-list marker is missing.

402 (NC_INVALID_FUNCTION)

The function code parameter is not in the supported range.

403 (NC_INVALID_POOL_NAME)

The pool selection parameter contains characters that are not allowed, or embedded spaces.

404 (NC_INVALID_COUNTER_NAME)

The *counter_name* parameter contains characters that are not allowed, or embedded spaces.

405 (NC_INVALID_VALUE_LENGTH)

The value length parameter is not in the range 0 to 8.

406 (NC_INVALID_COUNTER_VALUE)

The specified counter value or increment value is inconsistent with the minimum and maximum limits for the counter.

The counter value specified on the *current_value* parameter for the NC_CREATE function, or the *update_value* for the NC_UPDATE function, cannot be less than the specified minimum value, and cannot be more than (maximum value + 1).

The increment value specified in the *update_value* parameter for the NC_ASSIGN or NC_REWIND function cannot be greater than the total range of the counter ((maximum value – minimum value) + 1).

407 (NC_INVALID_COUNTER_LIMIT)

The maximum value is less than the minimum value.

408 (NC_INVALID_OPTIONS)

The value of the *counter_options* parameter is invalid. It is either a value that does not correspond to any defined option, or it is a value that represents some mutually exclusive options.

Named counter recovery

Named counters are only stored in the coupling facility. Applications using named counters might therefore need to implement recovery logic to handle the possible impact of any coupling facility problems.

If the coupling facility or list structure for the named counter pool fails, but another facility is available, it should normally be possible to recreate the named counter pool's list structure very quickly on another facility. The original instance of the server terminates as soon as it detects the problem, and a new instance is normally started immediately by ARM, provided that it is available, and the installation policy allows the restart. If the pool's list structure is known to have failed, the new server should be able to allocate a new instance immediately, and the pool should be available again within seconds.

However, in some situations, such as a coupling facility power failure, MVS might initially perceive the situation as a loss of connectivity, and be unable to determine that the list structure has failed until the original facility has been restarted. In such situations, recovery can be speeded up by issuing an operator command to force deletion of the existing structure, allowing a new instance to be allocated immediately.

Until the new structure has been created, attempts to obtain a counter value are rejected because the server is unavailable. This means that applications issuing such requests are unavailable while the new structure is being created, unless they have an alternative method of assigning numbers.

If the list structure for a named counter pool is recreated because of a failure, it is empty, and applications will immediately discover that their named counters are no longer found. The standard recovery technique in this situation is as follows:

1. Make the application issue an enqueue command (ENQ) for a resource based on the counter name. This ensures that only one task will be trying to repair each named counter.
2. Check to see if another task has already recreated the named counter.
3. If the named counter has not been recreated, recreate it with an appropriate value that you have reconstructed from other information, using the methods described below.
4. Issue a dequeue command (DEQ) for the resource, to release the named counter.

The enqueue and dequeue process is used in order to avoid multiple tasks wasting time and resources duplicating the same processing. However, if the process that you use to recreate the named counter is simple enough, you can omit the enqueue and dequeue process. If another task has already repaired the named counter, any subsequent attempt to recreate the counter will be rejected with a duplicate counter name indication.

If a named counter is only being used for temporary information that does not need to be preserved across a structure failure (for example, unique identifiers for active processes that would not be expected to survive such a failure), then recovery techniques can be minimal. For example, you could simply recreate the counter with a standard initial value.

However, if a named counter is being used for persistent information, such as an order number, recreating it may require specific application logic to reconstruct the

counter value. For example, the application could locate the highest key that is currently used in the order file. If active transactions might have already acquired new numbers from the counter, but not yet used them, then you should allow for this in the recovery process. Two methods of allowing for values that have been assigned, but not yet recorded, are:

1. Add a safety margin to the last used value, choosing a large enough margin so that the application should not set the counter to a value that might already have been assigned.
2. Treat all counter values as provisional. Restore the counter to the next apparently unused value, and in applications that use the counter, include logic to cover the situation where a counter value has already been assigned, and an application attempts to use it. The duplicate value can be detected by a duplicate key exception at the time the value is used (as a database or file key), at which point the application can obtain a new counter value and try again. Be careful to ensure that no side effects result from the original attempt to use the duplicated value.

The technique of locating the highest used counter value and provisionally assigning the next value can also be used as a backup method of assigning numbers when the named counter server is unavailable. However, it requires particularly careful verification and testing, because the logic to handle duplicate keys is normally only exercised in very unusual recovery situations.

If it is difficult to recreate the counter value from existing data repositories, then another possibility is that every so often (for example, once every 100 or 1000 numbers), the counter value could be logged to a record in a file. The recovery logic could recreate a suitable value for the named counter by taking the number logged in the file, and adding a safety margin, such as twice the interval at which the values are logged.

For systems running z/OS Release 3 or above, system-managed duplexing can be used to maintain duplexed copies of the named counters in different coupling facilities. This greatly reduces the risk of losing access to the counters, but it involves some cost in performance and resources. There is still some theoretical risk of losing the structure, perhaps because of operational errors or software problems, and any data in the coupling facility cannot be considered permanent, so some method of reconstructing counter values might still be required.

Part 5. Data Communication

Chapter 26. Terminal control.	365
Terminal access method support	366
Terminal control commands	366
Send/receive mode	367
Contention for the terminal	367
RETURN IMMEDIATE.	367
Speaking out of turn	368
Interrupting	369
Terminal waits	369
Using data transmission commands	370
What you get on a RECEIVE	370
Input chaining	370
Logical messages	370
NOTRUNCATE option	371
Print key	371
Device control commands	371
Terminal device support	372
Finding out about your terminal	375
EIB feedback on terminal control operations.	377
Using VTAM	378
Chaining input data	378
Chaining output data	378
Handling logical records	379
Response protocol	379
Using function management headers	380
Inbound FMH	380
Outbound FMH	380
Preventing interruptions (bracket protocol)	380
Using sequential terminal support	381
Coding considerations for sequential terminals	382
Print formatting	382
GOODNIGHT convention	382
Using TCAM	383
Coding for the TCAM/DCB interface.	383
Using batch data interchange	383
Destination selection and identification	385
Selection by named data set	385
Selection by medium	385
Definite response	385
Waiting for function completion	385
 Chapter 27. The 3270 family of terminals	 387
History of the 3270	387
Screen fields	388
Personal computers	388
PCs as 3270s	389
The 3270 buffer	390
The output datastream	390
3270 write commands	390
Write control character	391
3270 display fields	392
Display characteristics	392
3270 field attributes	392

Protection	393
Modification	393
Intensity	394
Base color	394
Extended attributes	394
Orders in the data stream	395
The start field order.	395
The modify field order.	396
The set buffer address order	397
The set attribute order.	398
Outbound data stream sample.	398
Input from a 3270 terminal	400
Data keys	400
Keyboard control keys.	401
Attention keys.	401
The AID	401
Reading from a 3270 terminal	403
Inbound field format	403
Input data stream example	404
Unformatted mode	404
 Chapter 28. CICS support for printing	 407
Formatting for CICS printers	407
CICS 3270 printers	408
CICS 3270 printer options	409
PRINT option and print control bit	409
ERASE option.	410
Line width options: L40, L64, L80, and HONEOM.	410
NLEOM option	410
Blank lines	411
Multiple sends.	411
Page width	411
Total page size	411
FORMFEED	411
PRINTERCOMP option	412
Non-3270 CICS printers	412
SCS input	413
Determining the characteristics of a CICS printer	413
BMS page size, 3270 printers	414
Supporting multiple printer types	414
Using CICS printers	415
Printing with a START command	415
Printing with transient data	415
Task that wants to print (on printer PRT1):	416
Task that gets triggered:	416
Printing with BMS routing	417
Using Non-CICS printers.	417
Formatting for non-CICS printers	417
Non-CICS printers: Delivering the data.	417
Programming for non-CICS printers	418
Notifying the print application	419
Printing display screens	420
CICS print key	420
ISSUE PRINT and ISSUE COPY.	420
Hardware print key	421
BMS screen copy	421

Chapter 29. CICS interface to JES	423
Using the CICS interface to JES	425
Spool interface restrictions	425
Creating output spool files	425
Using the MVS internal reader.	426
Reading input spool files	426
JES exits	427
Identifying spool files	427
Examples of SPOOL commands	430
COBOL	430
PL/I	431
C	431
ASSEMBLER	432
 Chapter 30. CICS intercommunication	 433
Design considerations	433
Programming language	434
Transaction routing	434
Function shipping	434
Distributed program link (DPL)	435
Using the distributed program link function	436
Examples of distributed program link	437
Programming considerations for distributed program link	442
Issuing multiple distributed program links from the same client task	442
Sharing resources between client and server programs	442
Mixing DPL and function shipping to the same CICS system	442
Mixing DPL and DTP to the same CICS system	443
Restricting a program to the distributed program link subset	443
Determining how a program was invoked.	443
Accessing user-related information with the ASSIGN command	443
Exception conditions for LINK command	444
Asynchronous processing	446
Distributed transaction processing (DTP)	446
Common Programming Interface Communications (CPI Communications)	446
External CICS interface (EXCI)	447

Chapter 26. Terminal control

The CICS application programming interface contains two sets of commands for communicating with terminals:

1. Terminal control commands
2. Basic Mapping Support (BMS)

Terminal control interface

Terminal control is the more basic of the two. It gives you flexibility and function, at the cost of more programming. In particular, if you code at the terminal control level, you need to build the device data stream in your application.

Terminal control commands apply to a variety of devices, reducing the sensitivity of programs to the terminals they support and to the access methods controlling the terminals. In addition to the commands themselves, CICS provides the data translation, synchronization of input and output operations, and session control needed to read from or write to a terminal or logical unit. This helps insulate you from the APIs of the individual communications access methods, which are complex and very different from one another.

BMS

BMS lets you communicate with a terminal at a much higher language level. It formats your data, and you do not need to know the details of the data stream. It is thus easier to code initially and easier to maintain, especially if your application has to support new types of terminal. However, BMS pathlengths are longer (BMS itself uses terminal control), and BMS does not support all the terminal types that terminal control does. BMS is described in Chapter 31, “Basic mapping support,” on page 453.

BMS insulates you even more from the characteristics of particular devices and the mechanics of communication than does terminal control, but at the cost of some flexibility and function.

This chapter describes:

- “Terminal access method support” on page 366
- “Terminal control commands” on page 366
- “Using data transmission commands” on page 370
- “Device control commands” on page 371
- “Terminal device support” on page 372
- “Finding out about your terminal” on page 375
- “Using VTAM” on page 378
- “Using sequential terminal support” on page 381
- “Using TCAM” on page 383
- “Using batch data interchange” on page 383

Terminal access method support

CICS Transaction Server for z/OS, Version 3 Release 1 supports terminals directly through interfaces to the following access methods:

Virtual Telecommunications Access Method (VTAM)

Basic Graphics Access Method (BGAM) for graphics terminals using GDDM®

Sequential Access Method (SAM) for terminals simulated by sequential devices

CICS supports operating system consoles as terminals too, but through operating system services rather than through an access method. The terminal control interface to a console is the same as to other terminals (though certain consoles might have certain restrictions), but BMS is not available. You can find a full list of the terminals supported by CICS in the *CICS Resource Definition Guide*.

Earlier releases of CICS also supported terminals through the 'DCB' interface of TCAM. You can still execute transactions under CICS from terminals using TCAM/DCB. However, the terminals themselves must be attached to a remote, pre-CICS TS 3.1, terminal-owning region. Local terminals using TCAM/DCB are not supported. The ACB interface of TCAM is not supported at all.

A transaction running under CICS communicates with a local surrogate for the remote TCAM terminal, and the two CICS systems manage the correspondence between the surrogate and the real terminal. The transaction is invoked either when the CICS that owns the terminal routes the transaction to the CICS region, or by automatic transaction initiation (ATI) in the CICS region. With ATI, this region arranges assignment of the terminal as principal facility for the transaction through the CICS region that owns the terminal.

Terminal control commands

The commands described in this chapter apply only to the principal facility of the task issuing them, where that facility is one of the following:

- A device connected through SAM
- An LU Type 0, 1, 2, 3, or 4 connected through VTAM.

Note: This chapter does not cover program-to-program communication, whether directed to the alternate or principal facility. This is covered in a separate manual, APPC commands are covered in the *CICS Distributed Transaction Programming Guide*.

Terminal control commands fall into four groups:

- Basic data transmission commands: RECEIVE, SEND, and CONVERSE
- Commands that send device controls, synchronize transmission, end a session, or perform similar control functions
- Commands to tell you about your terminal: ASSIGN and INQUIRE.
- Special device group commands: the batch data interchange (BDI) commands

Send/receive mode

The terminals and logical units covered in this chapter all operate in “half-duplex, flip-flop” mode. This means, essentially, that at any given moment, one partner in a conversation is in send mode (allowed to send data or control commands) and the other is in receive mode (restricted to receiving). This protocol is formally defined and enforced under VTAM. CICS observes the same conventions for terminals attached under other access methods, but both the hardware and the access methods work differently, so that not all operations are identical.

When a terminal is the principal facility of a task, its conversation partner is the task. When it is not associated with a task, its conversation partner is the terminal control component of CICS. Between tasks, under VTAM, the conversation is left in a neutral state where either partner can send first. Ordinarily the terminal goes first, sending the unsolicited input that initiates a task (see “How tasks are started” on page 142).

This transmission also reverses the send/receive roles; thereafter the terminal is in receive mode and CICS, represented by the task that was attached, is in send mode. The task starts and remains in send mode, no matter how many SENDs it executes, until it explicitly changes the direction of the conversation. One way in which you can put the task in receive mode is by specifying the INVITE option on a SEND command. After SEND with INVITE, the task is in receive mode and must issue a RECEIVE before sending to the terminal again. You can also put the task in receive mode simply by issuing a RECEIVE, without a preceding INVITE; INVITE simply optimizes transmissions.

Note that the first RECEIVE command in a task initiated by unsolicited input does not count in terms of send/receive mode, because the input message involved has long since been transmitted (it started the task). This RECEIVE just makes the message accessible to the task, and sets the related EIB fields.

ATI tasks—those initiated automatically by CICS—also start out in send mode, just like tasks started by unsolicited input.

Note that if a task is executing normally and performing non-terminal operations when a VTAM/network error occurs, the task is unaware of the error and continues processing until it attempts the next terminal control request. It is at this point that the task receives the TERMERR. If the task does not issue any further terminal control request, it will not receive the TERMERR or ABEND.

Contention for the terminal

CICS satisfies requests for automatic task initiation (ATI) as soon as the terminal required as principal facility is available. When a task ends at a terminal, and CICS has an ATI request for that terminal, there may be contention between CICS, which wants to initiate the ATI task, and the terminal user, who wants to initiate a certain task by unsolicited input. In this situation, CICS always sets itself up as contention *loser*. That is, if the terminal sends unsolicited input quickly enough after the end of the previous transaction, CICS creates a task to process it and delay fulfilling the ATI request. This is intentional—it gives the user priority in contention situations.

RETURN IMMEDIATE

However, you sometimes need to execute a sequence of particular tasks in succession at a terminal without allowing the user to intervene. CICS provides a way for you to do this, with the IMMEDIATE option on the RETURN command that ends the task. With RETURN IMMEDIATE, CICS initiates a task to execute the transaction named in the TRANSID option immediately, before honoring any other

waiting requests for tasks at that terminal and without accepting input from the terminal. The old task can even pass data to the new one. The new task accesses this data with a RECEIVE, as if the user had initiated the task with unsolicited input, but no input/output occurs. This RECEIVE, like the first one in a task initiated by unsolicited input, has no effect on send/receive status; it just makes the passed data available to the new task. If the terminal is using bracket protocol (explained in “Preventing interruptions (bracket protocol)” on page 380), CICS does not end the bracket at the end of the first task, as it ordinarily does, but instead continues the bracket to include the following task. Consequently, the automatic opening of the keyboard at the end of bracket between tasks does not occur.

Speaking out of turn

It is usually clear to users when they are supposed to “talk” (key and transmit), and when they are supposed to “listen” (wait for output), because the application makes this clear. On 3270 displays and many other terminals, the keyboard locks after the user has transmitted to reinforce this convention. It remains locked until the task unlocks it, which it usually does on a SEND before a RECEIVE, or on the last SEND in the task. This means the user has to do something particular (press the keyboard reset key) in order to break protocol.

What happens if the user does this? For terminals connected under VTAM, violating this protocol causes the task to abend (code ATCV) unless read-ahead queueing is in force. Read-ahead queueing allows the logical unit and the task to send and receive at any time; CICS saves input messages in temporary storage until the task needs them. Inputs not read by task end are discarded. Read-ahead queueing is applied at the transaction level (it is specified in the RAQ option of the PROFILE under which the transaction runs). Read-ahead queueing applies only to LU type 4 devices, and was originally provided for compatibility reasons, to allow a transaction to support both BTAM-connected and VTAM-connected terminals in the same way. As BTAM is no longer supported, read-ahead queueing should no longer be used.

Sequential terminals differ from others in send/receive rules. Because the input is a pre-prepared file, CICS simply provides input messages whenever the task requests them, and it is impossible to break protocol. If the input is improperly prepared, or is not what the task is programmed to handle, it is possible for the task to get out of synchronization with its inputs, to exhaust them prematurely, or to fail to read some of them.

Interrupting

VTAM provides a mechanism for a terminal in receive mode to tell its partner that it would like to send. This is the “signal” data flow in VTAM, which is detected on the next SEND, RECEIVE or ISSUE DISCONNECT command from the task. When a signal flow occurs, CICS raises the SIGNAL condition and sets EIBSIG in the EIB. CICS default action for the SIGNAL condition is to ignore it. For the signal to have any effect, the task must first detect the signal and then honor it by changing the direction of the conversation.

On a 3270 display terminal and some others, the ATTENTION key is the one that generates the interrupt. Not all terminals have this feature, however, and in VTAM, the bind image must indicate support for it as well, or VTAM ignores the interrupts.

Terminal waits

When a task issues a SEND command without specifying WAIT, CICS can defer transmission of the output to optimize either its overall terminal handling or the transmissions for your task. When it does this, CICS saves the output message and makes your task dispatchable, so that it can continue executing. The ISSUE COPY and ISSUE ERASE commands, which also transmit output, work similarly without WAIT.

If you use the WAIT option, CICS does not return control to your task until the output operation is complete. This wait lengthens the elapsed time of your task, with attendant effects on response time and memory occupancy, but it ensures that your task knows whether there has been an error on the SEND before continuing. You can avoid some of this wait and still check the completion of the operation if you have processing to do after your SEND. You issue the SEND without WAIT, continue processing, and then issue a WAIT TERMINAL command at the point where you need to know the results of your SEND.

When you issue a RECEIVE command that requires transmission of input, your task always waits, because the transmission must occur before the RECEIVE can be completed. However, there are cases where a RECEIVE does not correspond to terminal input/output. The first RECEIVE in a task initiated by unsolicited terminal input is the most frequent example of this, but there are others, as explained in the next section.

Also, when you issue any command involving your terminal, CICS ensures that the previous command is complete (this includes any deferred transmissions), before processing the new one.

Using data transmission commands

There are three commands that transmit data to and from the terminal or logical unit that is the principal facility of your task:

RECEIVE

reads data from the terminal.

SEND

writes data to the terminal.

CONVERSE

writes data to the terminal, waits for input, and reads the input.

CONVERSE is essentially a combination of SEND and RECEIVE and is usually the equivalent of SEND followed by RECEIVE. In certain cases you must use CONVERSE instead of SEND and RECEIVE, for example, sending structured-field data to certain 3270 devices. In other cases you must use SEND and RECEIVE, because CONVERSE is not provided; these are noted in Table 24 on page 374.

The SEND, RECEIVE, and CONVERSE commands are fully described in the *CICS Application Programming Reference* manual. They are broken down by device group, because the options for different devices and access methods vary considerably. “Terminal device support” on page 372 tells you which device group to use for your particular device.

What you get on a RECEIVE

We use the terms “input message” and “transmission” to mean both what the terminal sent and what the application received. For the most common types of terminals, these are equivalent. A 3270 display, for example, sends whatever was changed in its buffer as a single entity, and the task associated with the terminal normally gets the entire message in response to a single RECEIVE command.

However, input messages and physical transmissions are not always equivalent, and there are several factors that can affect the one-to-one relationship of either to RECEIVE commands. These are:

- VTAM chaining
- Logical records
- NOTRUNCATE option
- “Print” PA key

Input chaining

Some SNA devices break up long input messages into multiple physical transmissions, a process called “chaining”. CICS assembles the component transmissions into a single input message or present them individually, depending on how the terminal associated with the task has been defined. This affects how many RECEIVES you need to read a chained input message. Details on inbound chaining are explained in “Chaining input data” on page 378.

Logical messages

Just as some devices break long inputs into multiple transmissions, others block short inputs and send them in a single transmission. Here again, CICS provides an option about who deblocks, CICS or the receiving program. This choice also affects how much data you get on a single RECEIVE. (See “Handling logical records” on page 379 for more on this subject.)

NOTRUNCATE option

Still another exception to the one-input-message-per-RECEIVE rule occurs when the length of the input data is greater than the program expects. If this occurs and the RECEIVE command specifies NOTRUNCATE, CICS saves the excess data and uses it to satisfy subsequent RECEIVE commands from the program with no corresponding read to the terminal. If you are using NOTRUNCATE, you should issue RECEIVES until the field EIBCOMPL in the EIB is set on (that is, set to X'FF'). CICS turns on EIBCOMPL when no more of the input message is available.

Without NOTRUNCATE, CICS discards the excess data, turns on EIBCOMPL, and raises the LENGERR condition. It reports the true length of the data, before truncation, in the data area named in the LENGTH option, if you provide one.

Print key

If your CICS system has a PA key defined as a “print” key, another exception to the normal send/receive sequence can occur. If the task issues a RECEIVE, and the user presses the “print” key in response, CICS intercepts this input, does the necessary processing to fulfil the request, and puts the terminal in receive mode again. The user must send another input to satisfy the original RECEIVE. (See “CICS print key” on page 420 for more information about the “print” key.)

Device control commands

In addition to data transmission commands, the CICS API for terminals includes a series of commands that send instructions or control information, rather than data, to the terminal or to the access method controlling it. These commands are listed in the table below, along with a brief description of their function. Not all of these commands apply to all terminals, and for some, different forms apply to different terminals. See Terminal device support before going to the descriptions in the *CICS Application Programming Reference* manual .

The terminal in the table below is always the principal facility of the task issuing the command, except where explicitly stated otherwise. It may be a logical unit of a type not ordinarily considered a terminal.

Table 22. Control commands for terminals and logical units

Command	Action
FREE	Releases the terminal from the task, so that the terminal may be used in another task before the current one ends.
ISSUE COPY	Copies the buffer contents of the terminal named in the TERMDID option to the buffer of the terminal owned by the task. Both terminals must be 3270s.
ISSUE DISCONNECT	Schedules termination of the session between CICS and the terminal at the end of the task.
ISSUE EODS	Sends an end-of-data-set function management header (for 3650 interpreter logical units only).
ISSUE ERASEAUP	Erases all the unprotected fields of the terminal (for 3270 devices only).
ISSUE LOAD	Instructs the terminal to load the program named in the PROGRAM option (for 3650 interpreter logical units only).
ISSUE PASS	Schedules disconnection of the terminal from CICS and its transfer to the VTAM application named in the LUNAME option, at the end of the issuing task.

Table 22. Control commands for terminals and logical units (continued)

Command	Action
ISSUE PRINT	Copies the terminal buffer to the first printer eligible for a print request (for 3270 displays only).
WAIT SIGNAL	Suspends the issuing task until its terminal sends a SIGNAL dataflow command.
WAIT TERMINAL	Suspends the issuing task until the previous terminal operation has completed.

Terminal device support

Hardware and access method sensitivity is one of the major distinctions between using BMS and using terminal control commands to communicate with a terminal. BMS shields an application from hardware dependencies at the expense of some loss of function, whereas terminal control provides all the function.

The result of providing full function is that not all terminal control commands apply to all devices. Some commands require that you know what type of terminal you have, to determine the options that apply and the exceptional conditions that can occur. For some commands, you also need to know what access method is in use. The two tables that follow tell you which commands apply to which terminal and access method combinations. If you need to support several types of terminals, you can find out which type your task has as its principal facility using the commands described in “Finding out about your terminal” on page 375.

To use the tables, look up the terminal type that your program must support in the first column of Table 23. Use the value in the second column to find the corresponding command group in the first column of Table 24 on page 374. The second column of this table tells you the access method, and the third tells you the commands you can use. The commands themselves are described in full in the *CICS Application Programming Reference* manual. Where there is more than one version of a command in that manual, the table tells you which one to use. This information appears in parentheses after the command, just as it does in the manual itself.

Table 23. Devices supported by CICS

Device	Use commands for
2260, 2265	2260
3101 (supported as TWX 33/35)	3767
3230 (VTAM)	3767
3270 displays, 3270 printers (VTAM SNA)	LU type 2/3
3270 displays, 3270 printers (VTAM non-SNA)	3270 logical
3270 displays, 3270 printers (non-VTAM)	3270 display
SCS printers (VTAM)	SCS
3600 Pipeline mode (VTAM)	3600 pipeline
3601 (VTAM)	3600-3601
3614 (VTAM)	3600-3614
3630, attached as 3600 (3631, 3632, 3633, 3643, 3604)	Use 3600 entry
3641, 3644, 3646, 3647 (VTAM, attached as 3767)	3767

Table 23. Devices supported by CICS (continued)

Device	Use commands for
3643 (VTAM, attached as LU type 2)	LU type 2/3
3642, 3645 (VTAM, attached as SCS printer)	SCS
3650 interpreter LU	3650 interpreter
3650 host conversational LU (3270)	3650-3270
3650 host conversational LU (3653)	3650-3653
3650 host command LU (3680, 3684)	3650-3680
3650 interpreter LU	3650 interpreter
3650 host conversational LU (3270)	3650-3270
3650 host conversational LU (3653)	3650-3653
3650 host command LU (3680, 3684)	3650-3680
3730	3790 full function or inquiry
3767 interactive LU (VTAM)	3767
3770 Interactive LU (VTAM)	3767
3770 Full function LU	3790 full function or inquiry
3770 Batch LU (3771, 3773, 3774) (VTAM)	3770
3790 Full function or inquiry	3790 full function or inquiry
3790 3270 display LU	3790 3270-display
3790 SCS printer	3790 SCS
3790 3270 printer	3790 3270-printer
4700 (supported as 3600)	Use 3600 entry
5280 attached as 3270	Use 3270 entry
5520 VTAM, supported as 3790 full-function LU	3790 full function or inquiry
5550 (supported as 3270)	Use 3270 entry
5937 (supported as 3270)	Use 3270 entry
6670 VTAM	LU type 4
8130, 8140 under DPCX (supported as 3790)	3790 full function or inquiry
8100 DPPX/BASE using Host Presentation Services or Host Transaction Facility (attached as 3790)	3790 full function or inquiry
8100 DPPX/DSC, DPCX/DSC, including 8775 attach (supported as 3270)	LU type 2/3
8775	LU type 2/3
8815	APPC
Displaywriter supported as 3270	Use 3270 entry
Displaywriter supported as APPC	APPC
INTLU (interactive LU)	3767
PC, PS/2, attached as 3270	Use 3270 entry
Scanmaster	APPC
Series/1 supported as 3650 pipeline	3600 pipeline
Series/1 supported as 3790 full-function LU	3790 full function or inquiry
System/32 (5320) VTAM, supported as 3770	Use 3770 entry

Table 23. Devices supported by CICS (continued)

Device	Use commands for
System/34 (5340) VTAM, supported as 3770	Use 3770 entry
System/34 (5340) non-VTAM	System/3
System/36 (supported as System/34)	Use System/34 entry
System/38 (5381) VTAM, attached as 3770	Use 3770 entry
System/38 (5381) VTAM, attached as APPC	APPC
TWX 33/35 VTAM NTO	3767
WTTY VTAM NTO	3767

Table 24. Terminal control commands by device type

Device group name	Access methods	Commands applicable
2260	non-VTAM	RECEIVE (2260), SEND (2260), CONVERSE (2260), ISSUE DISCONNECT (default), ISSUE RESET
3270 display	non-VTAM	RECEIVE (3270 display), SEND (3270 display), CONVERSE (3270 display), ISSUE COPY (3270 display), ISSUE DISCONNECT (default), ISSUE ERASEAUP, ISSUE PRINT, ISSUE RESET
LU type 2/3 (3270 SNA)	VTAM	RECEIVE (LU type 2/3), SEND (LU type 2/3), CONVERSE (LU type 2/3), ISSUE COPY (3270 logical), ISSUE DISCONNECT (default), ISSUE ERASEAUP, ISSUE PASS, ISSUE PRINT
3270 logical (3270 non-SNA)	VTAM	RECEIVE (3270 logical), SEND (3270 logical), CONVERSE (3270 logical), ISSUE COPY (3270 logical), ISSUE DISCONNECT (default), ISSUE ERASEAUP, ISSUE PASS, ISSUE PRINT
SCS	VTAM	SEND (SCS), CONVERSE (SCS), ISSUE DISCONNECT (default), ISSUE PASS
3600 pipeline	VTAM	RECEIVE (3600 pipeline), SEND (3600 pipeline), ISSUE DISCONNECT (default), ISSUE PASS
3600-3601	VTAM	RECEIVE (3600-3601), SEND (3600-3601), CONVERSE (3600-3601), ISSUE DISCONNECT (default), ISSUE PASS, WAIT SIGNAL
3600-3614	VTAM	RECEIVE (3600-3614), SEND (3600-3614), CONVERSE (3600-3614), ISSUE DISCONNECT (default), ISSUE PASS
3650 interpreter	VTAM	RECEIVE (3650), SEND (3650 interpreter), CONVERSE (3650 interpreter), ISSUE DISCONNECT (default), ISSUE EODS, ISSUE LOAD, ISSUE PASS
3650-3270	VTAM	RECEIVE (3650), SEND (3650-3270), CONVERSE (3650-3270), ISSUE DISCONNECT (default), ISSUE ERASEAUP, ISSUE PASS, ISSUE PRINT
3650-3653	VTAM	RECEIVE (3650), SEND (3650-3653), CONVERSE (3650-3653), ISSUE DISCONNECT (default), ISSUE PASS

Table 24. Terminal control commands by device type (continued)

Device group name	Access methods	Commands applicable
3650-3680	VTAM	RECEIVE (3650), RECEIVE (3790 full function or inquiry), SEND (3650-3680), SEND (3790 full function or inquiry), CONVERSE(3650-3680), ISSUE DISCONNECT (default), ISSUE PASS
3767	VTAM	RECEIVE (3767), SEND (3767), CONVERSE (3767), ISSUE DISCONNECT (default), ISSUE PASS, WAIT SIGNAL
3770	VTAM	RECEIVE (3770), SEND (3770), CONVERSE (3770), ISSUE DISCONNECT (default), ISSUE PASS, WAIT SIGNAL
3790 full function or inquiry	VTAM	RECEIVE (3790 full function or inquiry), SEND (3790 full function or inquiry), CONVERSE (3790 full function or inquiry), ISSUE DISCONNECT (default), ISSUE PASS, WAIT SIGNAL
3790 3270-display	VTAM	RECEIVE (3790 3270-display), SEND (3790 3270-display), CONVERSE (3790 3270-display), ISSUE DISCONNECT (default), ISSUE ERASEAUP, ISSUE PASS, ISSUE PRINT
3790 3270-printer	VTAM	SEND (3790 3270-printer), ISSUE DISCONNECT (default), ISSUE ERASEAUP, ISSUE PASS
3790 SCS	VTAM	SEND (3790 SCS), ISSUE DISCONNECT (default), ISSUE PASS
LU type 4	VTAM	RECEIVE (LU type 4), SEND (LU type 4), CONVERSE (LU type 4), ISSUE DISCONNECT (default), ISSUE PASS, WAIT SIGNAL
Outboard controllers (batch data interchange)	VTAM	ISSUE ABORT, ISSUE ADD, ISSUE END, ISSUE ERASE, ISSUE NOTE, ISSUE QUERY, ISSUE RECEIVE, ISSUE REPLACE, ISSUE SEND, ISSUE WAIT
All others	VTAM	RECEIVE (VTAM default), SEND (VTAM default), CONVERSE (VTAM default), ISSUE PASS
All others	non-VTAM	RECEIVE (non-VTAM default), SEND (non-VTAM default), CONVERSE (non-VTAM default)

Finding out about your terminal

Some applications must support more than one type of terminal, and sometimes the types are sufficiently different that they require separate code. If you are writing such a program, and you need to determine what sort of terminal it is currently communicating with, you can use the ASSIGN command to find out.

ASSIGN returns a variety of information about the executing task, including a number of fields that describe its principal facility. Table 25 lists the ones that relate directly to terminal control operations. There are other ASSIGN options that relate to BMS and to other aspects of the task. You can find details on all ASSIGN options in the *CICS Application Programming Reference* manual. The “terminal” cited in column 2 of the table is always the principal facility of the task.

Table 25. ASSIGN command options for terminals

ASSIGN option	Information returned
ALTSCRNHT ALTSCRNWD	The alternate height and width of the terminal screen (from its terminal definition); see also SCRNHT and SCRNWD
APLKYBD	Whether terminal has an APL keyboard
APLTEXT	Whether terminal has the APL text feature
BTRANS	Whether terminal has background transparency capability
COLOR	Whether terminal has extended color capability
DEFSCRNHT DEFSCRNWD	The default height and width of the terminal screen (from its terminal definition); see also SCRNHT and SCRNWD
DELIMITER	The data-link control character for the terminal (for 3600 terminals only)
DESTID DESTIDLENGTH	The identifier of the outboard destination and its length (for BDI operations only)
DSSCS	Whether the terminal is an SCS data stream device
DS3270	Whether the terminal is a 3270 data stream device
EXTDS	Whether the terminal supports “query structured field” orders
EWASUPP	Whether the terminal supports “erase write alternate” orders (i.e. has alternate screen size capability)
FACILITY	The 4-character identifier of the terminal
FCI	The type of principal facility associated with the task (terminal, queue, and so on)
GCHARS GCODES	The graphic character set global identifier and the code page global identifier associated with the terminal
HILIGHT	Whether the terminal has extended highlight capability
KATAKANA	Whether the terminal supports Katakana
LANGINUSE	The 3-character mnemonic
MSRCONTROL	Whether the terminal supports magnetic slot reader control
NATLANGINUSE	The national language in use for the current task
NETNAME	The 8-character identifier of the terminal in the VTAM network
NUMTAB	Number of tabs required to position the print element in the correct passbook area (for 2980s only)
OPID OPCLASS	Operator identifier code and operator class of user signed on at terminal
OUTLINE	Whether the terminal has field outlining capability
PARTNS	Whether the terminal supports screen partitions
PS	Whether the terminal has programmed symbols capability
SCRNHT SCRNWD	Height and width of the terminal screen for the current task
SIGDATA	SIGNAL data received from the terminal
SOSI	Whether the terminal has mixed EBCDIC/double-byte character set capability
STATIONID TELLERID	Station and teller identifier of the terminal (for 2980s only)
TERMCODE	Type and model number of the terminal

Table 25. ASSIGN command options for terminals (continued)

ASSIGN option	Information returned
TERMPRIORITY	Terminal priority value
TEXTKYBD	Whether the terminal has the TEXTKYBD feature
TEXTPRINT	Whether the terminal has the TEXTPRINT feature
UNATTEND	Whether the terminal is unattended
USERID USERNAME USERPRIORITY	The 8-character identifier, 20-character name and priority of the user signed on at the terminal
VALIDATION	Whether the terminal has validation capability

You can also use the INQUIRE TERMINAL command to find out about your own terminal or any other terminal. INQUIRE TERMINAL returns information from the terminal definition, whereas ASSIGN describes the use of that terminal in the current task. For many options, however, particularly the hardware characteristics, the information returned is the same. INQUIRE TERMINAL is described in the *CICS System Programming Reference manual*.

EIB feedback on terminal control operations

CICS reports the results of processing terminal control commands, including those generated by BMS, in the EIB. Because of the complexity of terminal operations, many EIB fields are specific to terminal commands. Those that apply to the principal facility are listed in Table 26. (Other fields relate only to LU type 6.1, APPC and MRO operations; see the *CICS Application Programming Reference manual* for programming information about these.)

EIB fields are posted when CICS returns control to your task, and they always describe the most recent command to which they apply. This means that if you are conducting program-to-program communication over an alternate facility and using your principal facility, you need to check the EIB before results from one overlay those of the other.

It also means that when a task is initiated by unsolicited input from the terminal, or by a RETURN IMMEDIATE in the previous task at the same terminal, the EIB fields that describe the input are *not* set at task start. You must issue a RECEIVE to gain access to the input and post the EIB fields.

Note: If you are interested only in the EIB values and not the data itself, omit both the INTO and SET options from your RECEIVE.

Here are the fields that apply to the principal facility:

Table 26. EIB fields that apply to terminal control commands

Field	Contents
EIB AID	The attention identifier (AID) from the last input operation (3270s only, see “The AID” on page 401)
EIB ATT	Whether the input contains attach header data (an attach FMH)
EIB COMPL	Whether the RECEIVE command just issued used all the input data, or more RECEIVES are required (see “Chaining output data” on page 378)
EIB POSN	Cursor position at time of last input operation (3270s only)

Table 26. EIB fields that apply to terminal control commands (continued)

Field	Contents
EIBEOC	Whether an end-of-chain indicator appeared in the input from the last RECEIVE
EIBFMH	Whether user data just received or retrieved contains an FMH
EIBFREE	Whether the facility just used has been freed
EIBRCODE, EIBRESP, EIBRESP2	CICS response code values from the previously issued command Note: For output commands in which transmission can be deferred, these values reflect only the initial CICS processing of the command, not the eventual transmission (see “Terminal waits” on page 369).
EIBSIG	Whether the terminal has sent a SIGNAL
EIBTRMID	(CICS) identifier of the terminal

Using VTAM

Under VTAM, communication with logical units is governed by the conventions (protocols) which vary with the type of logical unit. This section describes the options provided by CICS to enable applications to conform to and make best use of these protocols,

Chaining input data

As noted earlier, some SNA devices segment long input messages for transmission. Each individual segment is called a **request unit** (RU), and the entire logical message is called a **chain**. CICS provides an option in the terminal definition, BUILDCHAIN, that governs who assembles the chain. If the BUILDCHAIN value for the terminal is YES, CICS assembles the chain and presents the entire message to the program in response to a single RECEIVE command. This choice ensures that the whole chain is complete and available before it is presented to the application.

If BUILDCHAIN=NO, the application assembles the chain. CICS provides one RU for each RECEIVE. The application can tell when it has received the last RU in the chain, because CICS raises the EOC (end-of-chain) condition at that time. CICS raises this condition even when there is only one RU in the chain, or when it assembles the chain, or when the input is from a terminal that does not support inbound chaining, like a 3270 display. An EOC condition is not considered an error; the CICS default action when it occurs is to ignore the condition.

EOC may occur simultaneously with either the EODS (end-of-data-set) or INBFMH (inbound-FMH) conditions, or both. Either condition takes precedence over EOC in determining where control goes if both it and EOC are the subject of active HANDLE CONDITION commands.

Chaining output data

VTAM supports the chaining of outbound as well as inbound terminal data. If the length of an output message exceeds the outbound RU size, and the terminal supports outbound chaining, CICS breaks the message into RU-size segments and transmits them separately.

Your application can take advantage of the fact that chaining is permitted by passing a single output message to CICS bit by bit across several SEND commands. To do this, you specify the CNOTCOMPL (“chain not complete”) option on each SEND except the one that completes the message. (Your message

segments do not have to be any particular length; CICS assembles and transmits as many RUs as are required.) The PROFILE definition under which your transaction is running must specify CHAINCONTROL=YES in order for you to do this.

Note: Options that apply to a complete logical message (that is, the whole chain) must appear only on the first SEND command for a chain. These include FMH, LAST, and, for the 3601, LDC.

Handling logical records

As noted earlier, some devices block input messages and send multiple inputs in a single transmission. CICS allows you to specify whether CICS or the application should deblock the input. The choice is expressed in the LOGREC option of the PROFILE under which the current transaction is executing.

With LOGREC (NO), CICS provides the entire input message in response to a RECEIVE (assuming the input is not chained or BUILDCHAIN=YES). The user is responsible for deblocking the input. If BUILDCHAIN=NO, a RECEIVE retrieves one RU of the chain at a time. In general, logical records do not span RUs, so that a single RU contains one or more complete logical records. The exception is LU type 4 devices, where a logical record may start in one RU and continue in another; for this reason, BUILDCHAIN=YES is recommended if you do your own deblocking for these devices.

If the PROFILE specifies LOGREC (YES), CICS provides one logical record in response to each RECEIVE command (whether or not CICS is assembling input chains).

If an RU contains more than one logical record, the records are separated by new line (NL) characters, X'15', interrecord separators (IRS characters), X'1E', or transparent (TRN) characters, X'35'. If NL characters are used, they are not removed when the data is passed to the program and appear at the end of the logical record. If IRS characters are used, however, they are removed. If the delimiter is a transparent character, the logical record can contain any characters, including NL and IRS, which are considered normal data in transparent mode. The terminating TRN is removed, however. CICS limits logical records separated by TRNs to 256 characters.

Response protocol

Under VTAM, CICS allows the use of either definite response or exception response protocol for outbound data.

Under exception response, a terminal acknowledges a SEND only if an error occurred. If your task is using exception response, CICS does not wait for the last SEND in the task (which may be the only SEND) to complete before terminating your task. Consequently, if an error does occur, it may not be possible to report it to your task. When this happens, the error is reported to a CICS-supplied task created for the purpose.

Definite response requires that the terminal acknowledge every SEND, and CICS does not terminate your task until it gets a response on the last SEND. Using definite response protocol has some performance disadvantages, but it may be necessary in some applications.

The MSGINTEG option of the PROFILE under which a task is running determines which response mode is used. However, if you select MSGINTEG (NO) (exception response), you can still ask for definite response on any particular SEND by using the DEFRESP option. In this way, you can use definite response selectively, paying the performance penalty only when necessary. For transactions that must verify the delivery of data before continuing, the DEFRESP option should be used on the last SEND.

Using function management headers

SNA architecture defines a particular type of header field that accompanies some messages, called a function management header (FMH). It conveys information about the message and how it should be handled. For some logical units, use of an FMH is mandatory, for others it is optional, and in some cases FMHs cannot be used at all. In particular, FMHs do *not* apply to LU type 2 and LU type 3 terminals, which are the most common 3270 devices.

Inbound FMH

When an FMH is present in an input message, CICS consults the PROFILE definition under which the transaction is executing to decide whether to remove it or pass it on to the application program that issued the RECEIVE. The PROFILE can specify that no FMHs are to be passed, that only the FMH indicating the end of the data set should be passed, or that all FMHs are to be passed. There is also an option that causes the FMH to be passed to the batch data interchange program.

If an FMH is present, it occupies the initial bytes of the input message; its length varies by device type. CICS sets the EIBFMH field in the EIB on (X'FF') to tell you that one is present, and it also raises the INBFMH condition, which you can detect through a HANDLE CONDITION command or by testing the RESP value.

Outbound FMH

On output, the FMH can be built by the application program or by CICS. If your program supplies the FMH, you place it at the front of your output data and specify the FMH option on your SEND command. If CICS is to build the FMH, you reserve the first three bytes of the message for CICS to fill in and omit the FMH option. CICS builds an FMH only for devices that require one; you must supply it for devices for which it is optional.

Preventing interruptions (bracket protocol)

Brackets are an SNA protocol for ensuring that a conversation between two LUs is not interrupted by a request from a third LU. CICS uses bracket protocol to prevent interruption of the conversation between a CICS task and its principal facility for the duration of the task. If the task has an alternate facility, bracket protocol is used there also, for the same reason. The logical unit begins the bracket if it sends unsolicited input to initiate the task, and CICS begins the bracket if it initiates the task automatically. CICS ends the bracket at task end, unless the IMMEDIATE option appears on the final RETURN command. RETURN IMMEDIATE lets you initiate another task at your principal facility without allowing it to enter input. CICS does this by *not* ending the bracket between the ending task and its successor when brackets are in use.

CICS requires the use of brackets for many devices under VTAM. For others, the use of brackets is determined by the value of the BRACKET option in the terminal definition. Because bracket protocol is a feature of SNA, if you specify BRACKET(YES) for non-SNA devices, CICS will neither follow, nor enforce, strict bracket protocol.

In general, bracket protocol is transparent to an application program, but it is still possible to optimize flows related to bracket protocol using the LAST option on the SEND command. If you know that a particular SEND is the last command for the terminal in a task, you can improve performance by adding the LAST option. LAST allows VTAM to send the “end-of-bracket” indicator with the data and saves a separate transmission to send it at task end. If you are sending the last output in a program-built chain (using CNOTCOMPL), LAST must be specified on the first SEND for the chain in order to be effective.

If your task has significant work to do or may experience a significant delay after its last SEND, you may want to issue a FREE command. FREE releases the terminal for use in another task.

Using sequential terminal support

One of the many types of terminal that CICS supports is not really a terminal at all, but a pair of sequential devices or files simulating a terminal. One of the pair represents the input side of the terminal, and might be a card reader, a spool file or a SAM file on tape or DASD. The other represents the output, and might be a printer, a punch, spool or SAM file. Many device-type combinations are allowed, and either of the pair can be missing; that is, you can have an input-only or output-only sequential terminal.

You read from and write to the devices or files that constitute a sequential terminal with terminal control commands, specifically RECEIVE, SEND, and CONVERSE. (BMS supports sequential terminals too; see “Special options for non-3270 terminals” on page 472.)

The original purpose of sequential terminal support was to permit application developers to test online code before they had access to real terminals. This requirement rarely occurs any more, but sequential terminals are still useful for:

Printing

See “Programming for non-CICS printers” on page 418. Sequential terminals are particularly useful for output that is sometimes directed to a low-speed CICS printer, for which BMS or terminal control commands are required, and sometimes directed to a high-speed system printer (spool or transient data commands). If you define the high-speed printer as a sequential terminal, you can use terminal control or BMS commands, and you can use the same code for both types of printers. (If there are differences in the device data streams, you need to use BMS for complete transparency.)

Regression testing

Tests run from sequential terminals leave a permanent record of both input and output. This encourages systematic and verifiable initial testing. Also, it allows you to repeat tests after modifications, to ensure that a given set of inputs produces the same set of outputs after the change as before.

Initialization

Some installations use a sequential terminal to execute one or more initialization transactions, in preference to program list table programs. Transactions initiated from a sequential terminal begin execution as soon as the terminal is in service, and they continue as quickly as CICS can process them until the input is exhausted. Hence the inputs from a sequential terminal can be processed immediately after startup, if the

sequential terminal is initially in service, at some later time (when it is put in service) or even as part of a controlled shutdown.

Coding considerations for sequential terminals

The input data submitted from a sequential terminal must be in the form in which it would come from a telecommunication device. For example, the first record usually starts with a transaction code, to tell CICS what transaction to execute. The transaction code must start in the first position of the input, just as it must on a real terminal. Note that this limits the ability to test applications that require input in complex formats. For example, there is no provision for expressing a formatted 3270 input stream as a sequential file, because of all the complex control sequences. However, you can use an unformatted 3270 data stream (or any other similar stream) for input, and you can still use BMS to format your output.

When you build the input file, you place an end-of-data indicator (EODI) character after each of your input records. The EODI character is defined in the system initialization table; the default value is a backslash ('\, X'E0'), but your installation may have defined some other value.

When processing the input stream, CICS observes EODI characters only. CICS
does not analyze the record structure of the input file or device, which means that
each input can span records in the input file. However, you must start each input on
a new physical record to ensure each input is correctly processed.

The length of an input record (the number of characters between EODIs) should not exceed the size of the input buffer (the INAREAL value in the LINE component of the sequential terminal definition). If it does, the transaction that attempts to RECEIVE the long record abends, and CICS positions the input file after the next EODI before resuming input processing.

An end-of-file marker in the input also acts as an EODI indicator. Any RECEIVE command issued after end-of-file is detected also causes an abend.

Print formatting

If the definition of a sequential terminal indicates that the output half is a line printer, you can write multiple lines of output with a single SEND. For this type of device, CICS breaks your output message into lines after each new line character (X'15') or after the number of characters defined as the line length, whichever occurs first. Line length is defined by the LPLEN value in the terminal definition. Each SEND begins a new line.

GOODNIGHT convention

CICS continues to initiate transactions from a sequential terminal until it (or the transactions themselves) have exhausted all the input or until the terminal goes out of service. To prevent CICS from attempting to read beyond the end of the input file (which causes a transaction abend), the last transaction executed can put the terminal out of service after its final output. Alternatively (and this is usually easier), the last input can be a CESF GOODNIGHT transaction, which signs the terminal off and puts it out of service. You cannot normally enter further input from a sequential terminal once CICS has processed its original input, without putting it out of service.

Using TCAM

Important

In CICS TS for z/OS, Version 3.1, local TCAM terminals are not supported. The only TCAM terminals supported are remote terminals connected to a pre-CICS TS 3.1 terminal-owning region by the DCB (not ACB) interface of TCAM. Thus, the only way in which TCAM is supported is by transaction routing or function shipping from a remote, pre-CICS TS 3.1, terminal-owning region, to which the terminals are connected by TCAM/DCB.

Coding for the TCAM/DCB interface

CICS does not support the DCB interface of TCAM directly. To use a terminal through this interface, you need to do so through an older version of CICS, as explained in “Terminal access method support” on page 366. In general, you use the same terminal control commands and options for such a device as you would if it were attached through VTAM. However, the path between your CICS application program and the terminal is much more complex, and consequently there are many more programming possibilities.

For programming information about the CICS-TCAM interface, see the *Customization Guide* for your older version of CICS (the terminal-owning region to which your TCAM terminals are connected).

Using batch data interchange

Many installations have a host computer and database at a central location, linked to other computers at branch offices. They do not necessarily contain CICS, but they can communicate with CICS in the host system. The CICS batch data interchange program provides for communication between an application program and a named data set (or destination) that is part of a batch data interchange logical unit in an outboard controller, or with a selected medium on a batch logical unit or an LU type 4 logical unit. This medium indicates the required device such as a printer or console.

The term “outboard controller” is a generalized reference to a programmable subsystem, such as the IBM 3770 Data Communication System, the IBM 3790 Data Communication System, or the IBM 8100 System running DPCX, which uses SNA protocols. (Details of SNA protocols and the data sets that can be used are given in *CICS/OS/VS IBM 3767/3770/6670 Guide* and *CICS/OS/VS IBM 3790/3780/8100 Guide*.) Figure 83 on page 384 gives an overview of batch data interchange.

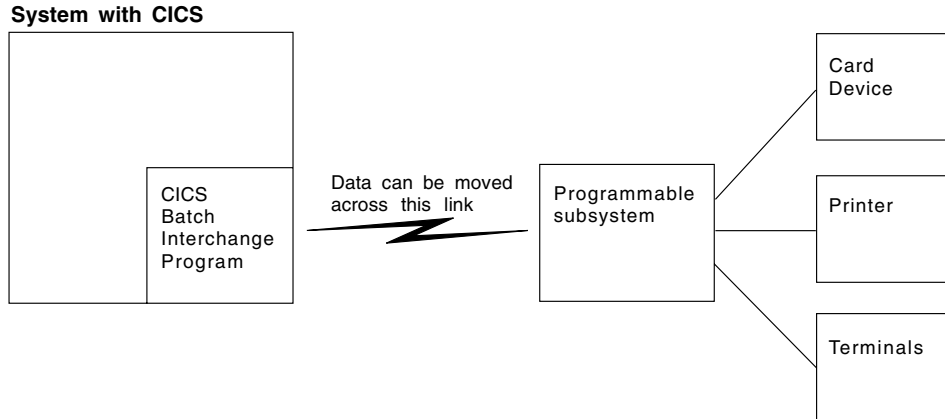


Figure 83. CICS batch data interchange

The following batch data interchange commands are provided:

ISSUE QUERY

Initiate transfer of a data set to the CICS application program.

ISSUE RECEIVE

Read a record from a data set or read data from an input medium.

ISSUE SEND

Transmit data to a named data set or to a selected medium.

ISSUE ADD

Add a record to a data set.

ISSUE REPLACE

Update (replace) a record in a data set.

ISSUE ERASE

Delete a record from a data set.

ISSUE END

Terminate processing of a data set.

ISSUE ABORT

Terminate processing of a data set abnormally.

ISSUE NOTE

Request the next record number in a data set.

ISSUE WAIT

Wait for an operation to be completed.

Where the controller is an LU type 4 logical unit, only the **ISSUE ABORT**, **ISSUE END**, **ISSUE RECEIVE**, **ISSUE SEND**, and **ISSUE WAIT** commands can be used.

Where the data set is a DPCX/DXAM data set, only the **ISSUE ADD**, **ISSUE ERASE**, and **ISSUE REPLACE** commands can be used.

Refer to Chapter 17, “Dealing with exception conditions,” on page 261 for information about how to deal with any exception conditions that occur during execution of a batch data interchange command.

Destination selection and identification

All batch data interchange commands except `ISSUE RECEIVE` include options that specify the destination. This is either a named data set in a batch data interchange logical unit, or a selected medium in a batch logical unit or LU type 4 logical unit.

Selection by named data set

The `DESTID` and `DESTIDLENG` options must always be specified, to supply the data set name and its length (up to a maximum of eight characters). For destinations having diskettes, the `VOLUME` and `VOLUMELENG` options may be specified, to supply a volume name and its length (up to a maximum of six characters); the volume name identifies the diskette that contains the data set to be used in the operation. If the `VOLUME` option is not specified for a multidiskette destination, all diskettes are searched until the required data set is found.

Selection by medium

As an alternative to naming a data set as the destination, various media can be specified by means of the `CONSOLE`, `PRINT`, `CARD`, or `WPMEDIA1–4` options. These media can be specified only in an `ISSUE ABORT`, `ISSUE END`, `ISSUE SEND`, or `ISSUE WAIT` command.

Definite response

CICS uses terminal control commands to carry out the functions specified in batch data interchange commands. For those commands that cause terminal control output requests to be made, the `DEFRESP` option can be specified. This option has the same effect as the `DEFRESP` option of the `SEND` terminal control command; that is, to request a definite response from the outboard controller, irrespective of the specification of message integrity for the CICS task (by the system programmer). The `DEFRESP` option can be specified for the `ISSUE ADD`, `ISSUE ERASE`, `ISSUE REPLACE`, and `ISSUE SEND` commands.

Waiting for function completion

For those batch data interchange commands that cause terminal control output requests to be made, the `NOWAIT` option can be specified. This option has the effect of allowing CICS task processing to continue; unless the `NOWAIT` option is specified, task activity is suspended until the batch data interchange command is completed. The `NOWAIT` option can be specified only on the `ISSUE ADD`, `ISSUE ERASE`, `ISSUE REPLACE`, and `ISSUE SEND` commands.

After a batch data interchange command with the `NOWAIT` option has been issued, task activity can be suspended, by the `ISSUE WAIT` command, at a suitable point in the program to wait for the command to be completed.

Chapter 27. The 3270 family of terminals

This chapter helps you to understand 3270 facilities and operation, so that you can use these terminals to best advantage in creating the end-user interface for your application. Some appreciation of the 3270 is also crucial to understanding BMS, because so many facilities of BMS exploit features of the 3270.

The 3270 is a family of display and printer terminals, with supporting control units, that share common characteristics and use the same encoded data format to communicate between terminal and host processor. This data format is known as the **3270 data stream**.

The 3270 is a complex device with many features and capabilities. Only basic operations are covered here and the emphasis is on the way CICS supports the 3270. For a comprehensive discussion of 3270 facilities, programming and data stream format, see the *IBM 3270 Information Display System Data Stream Programmer's Reference* manual. Programmers using terminal control commands still need to consult the *IBM 3270 Information Display System Data Stream Programmer's Reference* manual for details. The *IBM CICS/OS/VS 3270 Data Stream Device Guide* also contains much important information. It is primarily intended for programmers using terminal control, but contains information that may be helpful for BMS programmers as well. BMS support for a few special features is discussed in the BMS chapter. (See page Chapter 42, "Support for special hardware," on page 549 for more information.)

Although the discussion in this chapter is focused on display terminals, most of the material applies equally to 3270 printers. A 3270 printer accepts the same data stream as a 3270 display and simply delivers the screen image in hardcopy form. Most of the differences relate to input, which is (mostly) lacking on printers.

However, additional formatting facilities are available for use with printers, and there are special considerations in getting your printed output to the desired printer. For more information see Chapter 28, "CICS support for printing," on page 407.

This chapter describes:

- "History of the 3270"
- "The 3270 buffer" on page 390
- "The output datastream" on page 390
- "Orders in the data stream" on page 395
- "Outbound data stream sample" on page 398
- "Input from a 3270 terminal" on page 400
- "Inbound field format" on page 403
- "Unformatted mode" on page 404

History of the 3270

The development of the 3270 coincided with, and in part caused, the explosive growth of online transaction processing that began in the late 1960s. Consequently, the 3270 was a major influence in the design of transaction processing systems such as CICS.

The earliest terminal devices for online processing were adaptations of the teletype, the original and most basic computer terminal. Output was typed, and structure in the input typed by the operator was determined entirely by program convention, without any assists from the hardware. Cathode-ray tube terminals brought a

revolutionary improvement in output speed, allowing a complexity of application not previously possible, but formatting on early CRTs was not much more sophisticated than on their hard-copy predecessors.

Screen fields

The 3270 transformed the user interface by introducing the concept of **fields** on a display screen. Each field on the screen has a starting position and individual attributes, such as display intensity, color, and whether or not you can key data into it. Fields introduce structure into the communication between program and terminal operator in the same way that fields in a file record provide structure for interaction between programs and data.

```

Billing information on customer:
Reference Number      KRK123456
Full Name             Phileas Arthur Fogg
Amount Owed          $40.07
```

Figure 84. Part of a formatted screen, showing fields. Each block of text on the screen is a separate field. The fields on the left were filled in by program; those on the right were completed by an operator.

Organizing a screen display into fields has many advantages:

- The screen is easier to read, because fields can have different display characteristics.
- Data entry is enhanced by providing clear visual and keyboard cues about the order and format of the information required. The screen can be as explicit as a standard “fill-in-the-blanks” paper form. (Keyboard facilities reinforce the structure imposed by the fields. The keyboard locks if the operator tries to key into the wrong place. There are keys that tab from one field to the next, another that erases just the current field, and so on.)
- The length of the outbound data stream is reduced, because you send only nonblank (that is, nonspacer) data.
- The inbound data stream is also reduced, because the host normally reads only the changed fields.

Personal computers

The advent of personal computers (**PCs**) and intelligent workstations brought a second revolution in terminal display function. These terminals differ from 3270s in two important respects:

- They are generally “all points addressable”. That is, you can address any point on the display raster, just as you can on a television screen. A typical display might contain a grid of 640 by 480 points in the space normally used to display a single character on an earlier display. Moreover, a whole palette of colors and intensities is available at each point.

In contrast, a 3270 screen is divided into an array of character positions, typically 24 down and 80 across. Each position consists of an array of raster points, but you cannot address them individually. You can only select a character, from a set of about 190 choices, for each position. Some terminals allow you to select from several character sets and to load new sets, allowing a rudimentary form of graphics, but essentially you are working with a terminal that displays text,

numbers and symbols. You get some control of how the characters are displayed, but the choices are very limited in comparison with a PC display.

- The second difference is what makes the first possible. Personal computers and intelligent workstations contain a processor, memory, and programming (that is, “intelligence”) that make it possible to communicate with this very much more complex hardware through a relatively simple programming interface and minimum long-distance transmission of data.

These characteristics make possible a much higher-function end-user interface than that of the 3270. You can draw pictures, select from a variety of fonts, scale images in size, and so on. If you are writing a new application, and all of your users access it from such terminals, you may want to take advantage of this function to create the most efficient end-user interface possible for your application.

CICS cannot provide this type of function directly, but it does provide a number of ways for a task to communicate with a workstation, so that you can use a software package tailored for your particular workstation in combination with CICS. One popular approach is to use one of these packages, executing on the PC, to build your screens and handle the interactions with your user—that is, to implement the “front end” of your application. This code can then communicate with the part of your application that does the actual processing—the “back end” or “business logic” part—executing under CICS on the host. Communication between the two parts of the application can be done in several ways, depending on what your workstation supports:

- You can use one of the SNA application-to-application protocols, such as APPC.
- You can use the CPI-C “sockets” interface (see Chapter 30, “CICS intercommunication,” on page 433).
- You can use CICS on the workstation and use CICS facilities to communicate, or even distribute the business logic between the host and the workstation. CICS runs on many of these platforms, including OS/2, AIX®, OS/400®, and others.

When you do this, you can execute specific commands on the host (file operations, for example), or whole programs, or whole tasks. Executing commands remotely is called **function shipping**, executing a program remotely is called a **distributed program link**, and executing the whole task remotely is called **transaction routing**. See the *CICS Intercommunication Guide* for a full discussion of the possibilities, and the *CICS Distributed Transaction Programming Guide* for implementation details.

- You can use the terminal in emulation mode, a technique explained in “PCs as 3270s.”

If some of your users have 3270s or other nonprogrammable terminals, on the other hand, or if you are modifying an existing 3270 application, you need to use either terminal control or BMS commands.

PCs as 3270s

Although there is a different programming interface for a PC display, you can use PCs as “3270” terminals. Almost all PCs have programs available that emulate a 3270. These programs convert output in 3270 data stream format into the set of PC instructions that produces the same display on the screen, and similarly convert keyboard input into the form that would have come from a 3270 with the same screen contents.

Under an emulator, the PC display has essentially the same level of function as a real 3270. This limits your access to the more powerful PC hardware, although an emulator program often gives you a means to switch easily from its control to other

programs that use the display in full function mode. Moreover, the hardware on a particular PC does not always permit exact duplication of 3270 function (the keyboard may be different, for example). Consequently, your PC may not always behave precisely as described in this chapter or in the *IBM 3270 Information Display System Data Stream Programmer's Reference* manual, although the differences are usually minor.

The 3270 buffer

Communication with a 3270 device occurs through its **character buffer**, which is a hardware storage mechanism similar to the memory in a processor. Output to the 3270 is sent to the buffer. The buffer, in turn, drives the display of a display terminal and the print mechanism of a printer terminal.

Conversely, keyboard input reaches the host through the buffer, as explained in “Input from a 3270 terminal” on page 400.

Each position on the screen corresponds to one in the buffer, and the contents of that buffer position determine what is displayed on the screen. When the screen is formatted in fields, the first position of each field is used to store certain display characteristics of the field and is not available to display data (it appears blank). In the original models of the 3270, this byte was sufficient to store all of the display characteristics. In later models, which have more types of display characteristics, the additional information is kept in an area of buffer storage not associated with a fixed position on the screen. See “Display characteristics” on page 392 for more about display characteristics.

The output datastream

To create a 3270 display, you send a stream of data that consists of:

- A write command (one byte)
- A write control character or **WCC** (one byte)
- Display data (variable number of bytes)

The WCC and display data are not always present; the write command determines whether a WCC follows and whether data may or must be present.

When you use BMS, CICS builds the entire data stream for you. The WCC is assembled from options in the SEND command, and the write command is selected from other SEND options and information in the PROFILE of the transaction being executed. Display data is built from map or text data that you provide, which BMS translates into 3270 format for you.

When you use terminal control commands, such as SEND, CICS still supplies the write command, built from the same information. However, you provide the WCC and you must express the display data in 3270 format.

3270 write commands

Even though CICS supplies the write command, you need to know the possibilities, so that you can select the options that produce the one you want. There are five 3270 commands that send data or instructions to a terminal:

- Write
- Erase/write
- Erase/write alternate
- Erase all unprotected fields

- Write structured fields

The 3270 **write** command sends the data that follows it to the 3270 buffer, from which the screen (or printer) is driven. **Erase/write** and **erase/write alternate** also do this, but they erase the buffer first (that is, they set it entirely to null values). They also determine the buffer size (the number of rows and columns on the screen), if the terminal has a feature called **alternate screen size**.

Terminals with this feature have two sizes, **default size** and **alternate size**. The erase/write command causes the default size to be used in subsequent operations (until the next erase/write or erase/write alternate command), and erase/write alternate selects the alternate size, as the names suggest.

CICS uses the plain write command to send data unless you include the ERASE option on your SEND command. If you specify ERASE DEFAULT on your SEND, CICS uses erase/write instead (setting the screen to default size), and ERASE ALTERNATE causes CICS to use erase/write alternate (setting alternate size). If you specify ERASE without DEFAULT or ALTERNATE, CICS looks at the PROFILE definition associated with the transaction you are executing to decide whether to use erase/write or erase/write alternate.

The **erase unprotected to address** command causes a scan of the buffer for unprotected fields (these are defined more precisely in “3270 field attributes” on page 392). Any such fields that are found are set to nulls. This selective erasing is useful in data entry operations, as explained in “The SEND CONTROL command” on page 485. No WCC or data follows this command; you send only the command.

Write structured fields causes the data that follows to be interpreted as 3270 structured fields. **Structured fields** are required for some of the advanced function features of the 3270. They are not covered here, but you can write them with terminal control SEND commands containing the STRFIELD option. See the *IBM CICS/OS/VS 3270 Data Stream Device Guide* if you wish to do this.

Write control character

The byte that follows a 3270 write, erase/write or erase/write alternate command is the **write control character** or **WCC**. The WCC tells the 3270 whether or not to:

- Sound the audible alarm
- Unlock the keyboard
- Turn off the modified data tags
- Begin printing (if terminal is a printer)
- Reset structured fields
- Reset inbound reply mode

In BMS, CICS creates the WCC from the ALARM, FREEKB, FRSET, and PRINT options on your SEND MAP command. If you use terminal control commands, you can specify your WCC explicitly, using the CTLCHAR option. If you do not, CICS generates one that unlocks the keyboard and turns off the modified data tags (these are explained shortly, in “Modification” on page 393).

3270 display fields

Display data consists of a combination of characters to be displayed and instructions to the device on how and where to display them. Under ordinary circumstances, this data consists of a series of field definitions, although it is possible to write the screen without defining fields, as explained in “Unformatted mode” on page 404.

After a write command that erases, you need to define every field on the screen. Thereafter, you can use a plain write command and send only the fields you want to change.

To define a field, you need to tell the 3270:

- How to display it
- What its contents are
- Where it goes on the screen (that is, its starting position in the buffer)

Display characteristics

Each field on the screen has a set of display characteristics, called **attributes**. Attributes tell the 3270 *how* to display a field, and you need to understand what the possibilities are whether you are using BMS or terminal control commands. Attributes fall into two categories:

Field attributes

These include:

- Protection (whether the operator can modify the field or not)
- Modification (whether the operator *did* modify the field)
- Display intensity

All 3270s support field attributes; “3270 field attributes” explains your choices for them.

Field attributes are stored in the first character position of a field. This byte takes up a position on the screen and not only stores the field attributes, but marks the beginning of the field. The field continues up to the next attributes byte (that is, to the beginning of the next field). If the next field does not start on the same line, the current one wraps from the end of the current line to the beginning of the next line until another field is encountered. A field that has not ended by the last line returns to the first.

Extended field attributes

(Usually shortened to **extended attributes**). These are not present on all models of the 3270. Consequently, you need to be aware of which ones are available when you design your end-user interface. Extended attributes include special forms of highlighting and outlining, the ability to use multiple symbol sets and provision for double-byte character sets. Table 27 on page 394 lists the seven extended attributes and the values they can take.

3270 field attributes

As noted above, the field attributes byte holds the protection, modification and display intensity attributes of a field. Your choices for each of these attributes are described here using the terms that BMS uses in defining formats. If you use terminal control commands, you need to set the corresponding bits in the attributes byte to reflect the value you choose.

(See the *IBM 3270 Information Display System Data Stream Programmer's Reference* manual for the bit assignments. See also "Attribute value definitions: DFHBMSCA" on page 481 for help from CICS in this area.)

Protection

There are four choices for the protection attribute, using up two bit positions in the attributes byte. They are:

Unprotected

The operator can enter any data character into an unprotected field.

Numeric-only

The effect of this designation depends on the keyboard type of the terminal. On a data entry keyboard, a numeric shift occurs, so that the operator can key numbers without shifting. On keyboards equipped with the "numeric lock" special feature, the keyboard locks if the operator uses any key except one of the digits 0 through 9, a period (decimal point), a dash (minus sign) or the DUP key. This prevents the operator from keying alphabetic data into the field, although the receiving program must still inspect the entry to ensure that it is a number of the form it expects. Without the numeric lock feature, numeric-only allows any data into the field.

Protected

The operator cannot key into a protected field. Attempting to do so locks the keyboard.

Autoskip

The operator cannot key into an autoskip field either, but the cursor behaves differently. (The cursor indicates where the operator's next keystroke will go; for more information about this, see "Input from a 3270 terminal" on page 400.) Whenever the cursor is being advanced to a new field (either because the previous field filled or because a field advance key was used), the cursor skips over any autoskip fields in its path and goes to the first field that is either unprotected or numeric-only.

Modification

The second item of information in the field attributes byte occupies only a single bit, called the **modified data tag** or **MDT**. The MDT indicates whether the field has been modified or not. The hardware turns on this bit automatically whenever the operator makes any change to the field contents. The MDT bit is very important because, for the read command that CICS normally uses, it determines whether the field is included in the inbound data or not. If the bit is on (that is, the field was changed), the 3270 sends the field; if not, the field is not sent.

You can also turn the MDT on by program, when you send a field to the screen. Using this feature ensures that a field is returned on a read, even if the operator cannot or does not change it. The FRSET option on BMS SEND commands allows you to turn off the tags for all the fields on the screen by program; you cannot turn off individual tags by program. If you are using terminal control commands, you turn on a bit in the WCC to turn off an individual tag.

Intensity

The third characteristic stored in the attributes byte is the display intensity of the field. There are three mutually exclusive choices:

Normal intensity

The field is displayed at normal brightness for the device.

Bright The field is displayed at higher than normal intensity, so that it appears highlighted.

Nondisplay

The field is not displayed at all. The field may contain data in the buffer, and the operator can key into it (provided it is not protected or autoskip), but the data is not visible on the screen.

Two bits are used for display intensity, which allows one more value to be expressed than the three listed above. For terminals that have either of the associated special hardware features, these same two bits are used to determine whether a field is light-pen detectable or cursor selectable. Because there are only two bits, not all combinations of intensity and selectability are possible. The compromise is that bright fields are always detectable, nondisplay fields are never detectable, and normal intensity fields may be either. "Cursor and pen-detectable fields" on page 552 contains more information about these features.

Base color

Some terminals support **base color** without, or in addition to, the **extended colors** included in the extended attributes. There is a mode switch on the front of such a terminal, allowing the operator to select base or default color. Default color shows characters in green unless field attributes specify bright intensity, in which case they are white. In base color mode, the protection and intensity bits are used in combination to select among four colors: normally white, red, blue, and green; the protection bits retain their protection functions as well as determining color. (If you use extended color, rather than base color, for 3270 terminals, note that you cannot specify "white" as a color. You need to specify "neutral", which is displayed as white on a terminal.)

Extended attributes

In addition to the field attributes just described, some 3270 terminals have extended attributes as well. Table 27 lists the types of extended attributes in the first column and the possible values for each type in the second column.

Table 27. 3270 extended attributes

Attribute type	Values
Extended color	Blue, red, pink, green, turquoise, yellow, neutral
Extended highlighting	Blinking, reverse video, underscoring
Field outlining	Lines over, under, left and right, in any combination
Background transparency	Background transparent, background opaque
Field validation	Field must be entered; field must be filled; field triggers input

Table 27. 3270 extended attributes (continued)

Attribute type	Values
Programmed symbol sets	Number identifying the symbol set Note: The control unit associated with a terminal contains a default symbol set and can store up to five additional ones. To use one of these others, you need to load the symbol set into the controller prior to use. You can use a terminal control SEND command to do this.
SO/SI creation	Shift characters indicating double-byte characters may be present; shift characters are not present

The *IBM 3270 Information Display System Data Stream Programmer's Reference* manual contains details about extended attributes and explains how default values are determined. You can use ASSIGN and INQUIRE commands to determine which extended attributes your particular terminal has. These commands are described in "Finding out about your terminal" on page 375.

Some models of the 3270 also allow you to assign extended attribute values to individual characters within a field that are different from the value for the field as a whole. Generally, you need to use terminal control commands to do this, because BMS does not make explicit provision for character attributes. However, you can insert the control sequences for character attributes in text output under BMS, as explained in "Text lines" on page 523. "The set attribute order" on page 398 describes the format of such a sequence.

Orders in the data stream

The next several sections tell you how to format outbound data to express the attributes, position, and contents of a field. You need to know this information if you are writing to a 3270 using terminal control commands. If you are using BMS, all this is done for you, and you can move on to "Input from a 3270 terminal" on page 400.

When you define a field in the 3270 data stream, you begin with a **start field (SF)** or a **start field extended (SFE)** order. **Orders** are instructions to the 3270. They tell it how to load its buffer. They are one byte long and usually are followed by data in a format specific to the order.

The start field order

The SF order is supported on all models and lets you specify the field attributes and the display contents of a field, but not extended attributes. To define a field with SF, you insert a sequence in the data stream as in Figure 85 on page 396.

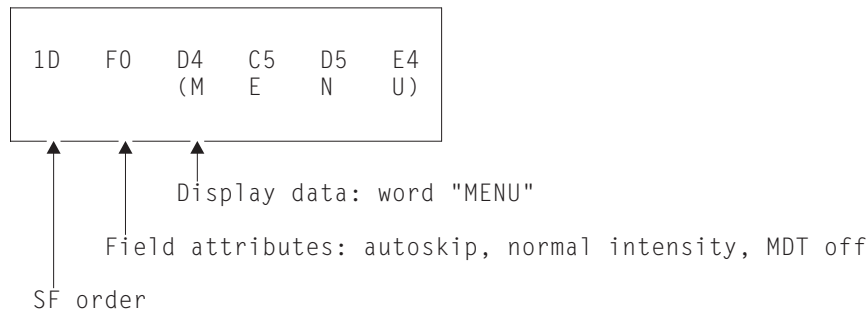


Figure 85. Field definition using SF order

If you need to specify extended attributes, and your terminal supports them, you use the start field extended order instead. SFE requires a different format, because of the more complex attribute information. Extended attributes are expressed as byte pairs. The first byte is a code indicating which type of attribute is being defined, and the second byte is the value for that attribute. The field attributes are treated collectively as an additional attribute type and also expressed as a byte pair. Immediately after the SFE order, you give a 1-byte count of the attribute pairs, then the attribute pairs, and finally the display data. The whole sequence is shown in Figure 86.

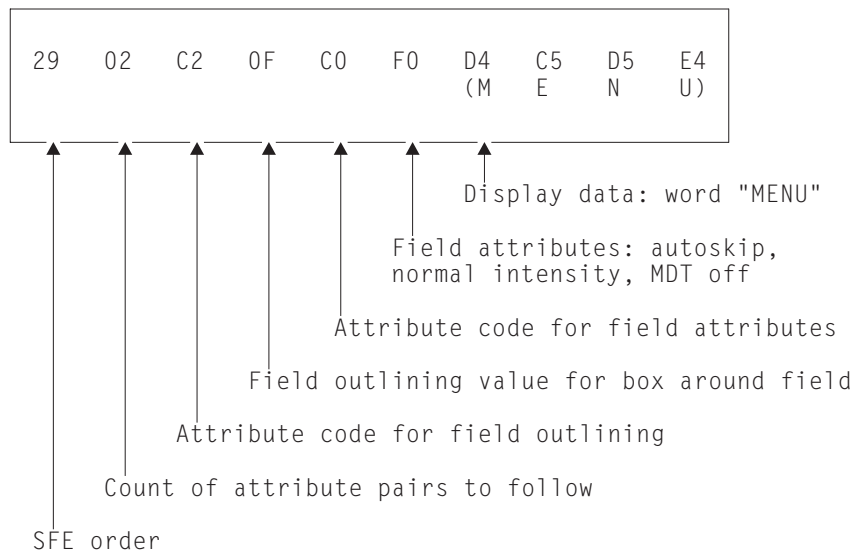


Figure 86. Field definition using SFE order

The modify field order

When a field is on the screen, you can change it with a command almost identical in format to SFE, called **modify field (MF)**. The only differences from SFE are:

- The field must already exist.
- The command code is X'2C' instead of X'29'.
- You send only the attributes you want to change from their current values, and you send display data only if you want to change it.
- A null value sets an attribute back to its default for your particular terminal (you accomplish the same thing in an SFE order by omitting the attribute).

For example, to change the “menu” field of earlier examples back to the default color for the terminal and underscore it, you would need the sequence in Figure 87.

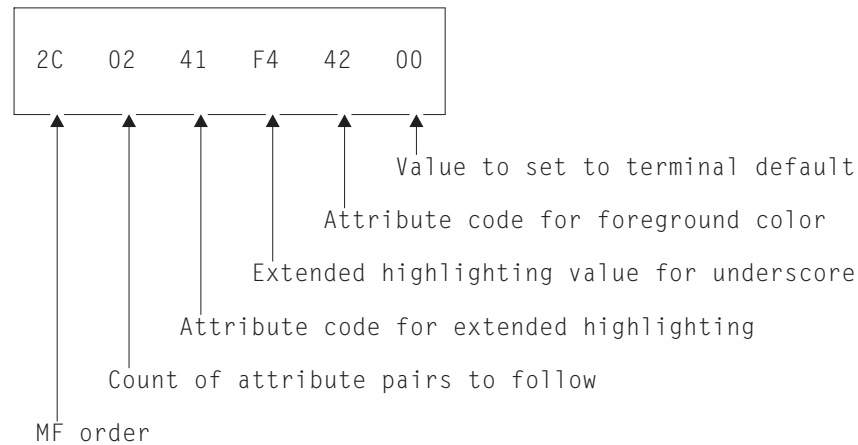


Figure 87. Changing field attributes within an MF order

The set buffer address order

The SF and SFE orders place the field they define at the current position in the buffer, and MF modifies the field at this position. Unless the field follows the last character sent (that is, begins in the current buffer position), you need to precede these orders with a **set buffer address (SBA)** order to indicate where you want to place or change a field. To do this, you send an SBA order followed by a 2-byte address, as in Figure 88.



Figure 88. SBA sequence

The address in the figure is a “12-bit” address for position 112 (X'70'), which is row 2, column 33 on an 80-column screen. Note that counting starts in the first row and column (the zero position) and proceeds along the rows. There are two other addressing schemes used: “14-bit” and “16-bit”. Buffer positions are numbered sequentially in all of them, but in 12- and 14-bit addressing, not all the bits in the address are used, so that they do not appear sequential. (The X'70' (B'1110000') in the figure appears as B'110000' in the low-order six bits of the rightmost byte of the address and B'000001' in the low-order six bits of the left byte.) The *IBM 3270 Information Display System Data Stream Programmer's Reference* manual explains how to form addresses.

After an SF, SFE, or MF order, the current buffer address points to the first position in the buffer you did not fill—right after your data, if any, or after the field attributes byte if none.

The set attribute order

To set the attributes of a single character position, you use a **set attribute (SA)** order for each attribute you want to specify. For example, to make a character blink, you need the sequence in Figure 89.

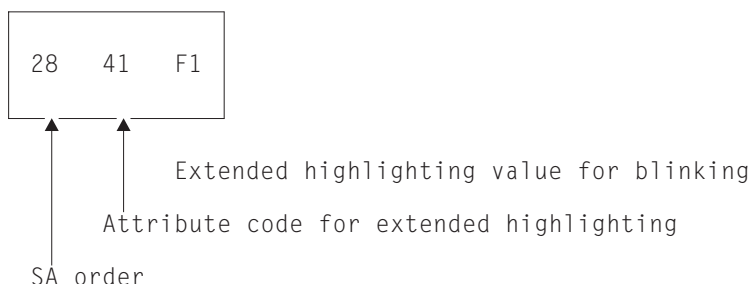


Figure 89. SA sequence to make a character blink

The attributes you specify with SA orders are assigned to the current buffer position, in the same way that field definitions are placed at the current buffer position, so you generally need to precede your SAs with SBA sequences.

Outbound data stream sample

This section shows you an annotated example of the data stream required to paint a particular 3270 screen, to reinforce the explanation of how the data stream is built.

Figure 90 shows an example screen that is part of an application that keeps track of cars used by the employees at a work site, and is used to record a new car. The only inputs are the employee identification number, the license plate (tag) number, and, if the car is from out-of-state, the licensing state.

Car Record		
Employee No: _____	Tag No: _____	State: ____

Figure 90. Example of a data-entry screen

Note: This is an unrealistically simple screen, designed to keep the explanation manageably short. It does not conform to generally accepted standards of screen design, and you should not use it as a model.

There are eight fields on this screen:

1. Screen title, "Car Record", on line 1, column 26
2. Label field, "Employee No:" (line 3, column 1), indicating what the operator is to enter into the next field
3. An input field for the employee number (line 3, column 14), six positions long
4. Label field, "Tag. No:", at line 3, column 21
5. An input field (tag number) at line 3, column 31, eight positions long
6. Label field, "State:", at line 3, column 40
7. An input field (state), at line 3, column 49, two positions long
8. A field to mark the end of the previous (state) input field, at line 3, column 52

Table 28 shows the outbound data stream:

Table 28. 3270 output data stream

Bytes	Contents	Notes
1	X'F5'	The 3270 command that starts the data stream, in this case erase/write.
2	X'C2'	WCC; this value unlocks the keyboard, but does not sound the alarm or reset the MDTs.
3	X'11'	SBA order to position first field at ...
4-5	X'40D6'	Address of line 1, column 23 on 24 by 80 screen, using 12-bit addressing.
6	X'1D'	SF order to begin first field definition.
7	X'F8'	Field attributes byte; this combination indicates a field which is autoskip and bright, with the MDT initially off.
8-17	'Car record'	Display contents of the field.
18-20	X'11C260'	SBA sequence to reset the current buffer position to line 3, column 1 for second field.
21	X'1D'	SF order for second field.
22	X'F0'	Field attributes byte: autoskip, normal intensity, MDT off.
23-34	'Employee No.'	Display contents of field.
35	X'29'	SFE order to start fourth field. SFE is required, instead of SF, because you need to specify extended attributes. This field starts immediately after the previous one left off, so you do not have to precede it with an SBA sequence.
36	X'02'	Count of attribute <i>types</i> that are specified (two here: field outlining and field attributes).
37	X'41'	Code indicating attribute type of extended highlighting.
38	X'F4'	Extended highlighting value indicating underscoring.
39	X'C0'	Code indicating attribute type of field attributes.
40	X'50'	Field attributes value for numeric-only, normal intensity, MDT off. Any initial data for this field would appear next, but there is none.
41	X'13'	Insert cursor (IC) order, which tells the 3270 to place the cursor at the current buffer position. We want it at the start of the first field which the operator has to fill in, which is the current buffer position.
42-44	X'11C2F4'	SBA sequence to position to line 3, column 21, to leave the six positions required for an employee number. The beginning of the "Tag No" label field marks the end of the employee number input field, so that the user is aware immediately if he tries to key too long a number.
45	X'1D'	SF order to start field.
46	X'F0'	Field attributes byte: autoskip, normal intensity, MDT off.
47-55	' Tag No.'	Display data. We attach two leading blanks to the label for more space between the fields. (We could have used a separate field, but this is easier for only a few characters.)
56	X'29'	SFE (the next field is another input field, where we want field outlining, so we use SFE again).

Table 28. 3270 output data stream (continued)

Bytes	Contents	Notes
57	X'02'	Count of attribute types.
58-59	X'41F4'	Code for extended highlighting with value of underscoring.
60-61	X'C040'	Code for field attributes and attributes of unprotected, normal intensity, MDT off.
62-64	X'11C3C7'	SBA sequence to reposition to line 3, column 40, leaving eight positions for the tag.
65	X'1D'	SF to start field.
66	X'F0'	Field attributes byte: autoskip, normal intensity, MDT off.
67-74	' State:'	Field data (two leading blanks again for spacing).
75-80	X'290241F4C040'	SFE order and attribute specifications for state input field (attributes are identical to those for tag input field).
81-82	X'0000'	The (initial) contents of the state field. We could have omitted this value as we did for other input fields, but we would need an SBA sequence to move the current buffer position to the end of the field, and this is shorter.
83	X'1D'	SF. The last field indicates the end of the previous one, so that the user does not attempt to key more than two characters for the state code. It has no initial data, just an attributes byte. This kind of field is sometimes called a "stopper" field.
84	X'F0'	Field attributes byte: autoskip, normal intensity, MDT off.

Note: If you use terminal control commands and build your own data stream, the data you provide in the FROM parameter of your SEND command starts at byte 3 in the table above; CICS supplies the write command and the WCC from options on your SEND command.

Input from a 3270 terminal

As explained earlier, keyboard input reaches the host through the buffer. There are many different keyboard arrangements available for 3270 terminals, but in any arrangement, a key falls into one of three categories:

- Data key
- Keyboard control key
- Attention key

Data keys

The data keys include all the familiar letters, numbers, punctuation marks and special characters. Depressing a data key simply changes the content of the buffer (and therefore the screen) at the point indicated by the **cursor**. The cursor is a visible pointer to the position on the screen (that is, in the buffer) where the next data keystroke is to be stored. As the operator keys data, the cursor advances to the next position on the screen, skipping over fields defined with the autoskip attribute on the screens that have been formatted.

Keyboard control keys

Keyboard control keys move the cursor to a new position, erase fields or individual buffer positions, cause characters to be inserted, or otherwise change where or how the keyboard modifies the buffer.

Attention keys

The keys in the previous groups, Data and Keyboard control keys, cause no interaction with the host; they are handled entirely by the device and its control unit. An attention key, on the other hand, signals that the buffer is ready for transmission to the host. If the host has issued a read to the terminal, the usual situation in CICS, transmission occurs at this time.

There are five types of attention key:

- ENTER
- PF (program function) key
- CLEAR
- PA (program attention) key
- CNCL (cancel key, present only on some keyboard models)

In addition to pressing an attention key, there are other operator actions that cause transmission:

- Using an identification card reader
- Using a magnetic slot reader or hand scanner
- Selecting an attention field with a light pen or the cursor select key
- Moving the cursor out of a trigger field

Trigger field capability is provided with extended attributes on some terminal models, but all the other actions listed above require special hardware, and in most cases the screen (buffer) must be set up appropriately beforehand. We talk about these features in Chapter 42, “Support for special hardware,” on page 549. For this chapter, we concentrate on standard features.

The AID

The 3270 identifies the key that causes transmission by an encoded value in the first byte of the inbound data stream. This value is called the **attention identifier** or **AID**.

Ordinarily, the key that the terminal operator chooses to transmit data is dictated by the application designer. The designer assigns specific meanings to the various attention keys, and the user must know these meanings in order to use the application. (Often, there are only a few such keys in use: ENTER for normal inputs, one PF key to exit from control of the application, another to cancel a partially completed transaction sequence, for example. Where there are a number of choices, you may want to list the key definitions on the screen, so that the user does not have to memorize them.)

There is an important distinction between two groups of attention keys, which the application designer must keep in mind. The ENTER and PF keys transmit data from the buffer when the host issues a “read modified” command, the command normally used by CICS. CLEAR, CNCL and the PA keys do not, although you do get the AID (that is, the identity of the key that was used). These are called the **short read** keys. They are useful for conveying simple requests, such as “cancel”, but not for those that require accompanying data. In practice, many designers use PF keys even for the nondata requests, and discard any accompanying data.

Note: The CLEAR key has the additional effect of setting the entire buffer to nulls, so that there is literally no data to send. CLEAR also sets the screen size to the default value, if the terminal has the alternate screen size feature, and it puts the screen into unformatted mode, as explained in “Unformatted mode” on page 404.

Reading from a 3270 terminal

There are two basic read commands for the 3270:

- **Read buffer**
- **Read modified**

For either command, the inbound data stream starts with a 3-byte **read header** consisting of:

- Attention identifier (AID), one byte
- Cursor address, two bytes

As noted in the previous section, the AID indicates which action or attention key causes transmission. The cursor address indicates where the cursor was at the time of transmission. CICS stores this information in the EIB, at EIBAID and EIBCPOSN, on the completion of any RECEIVE command.

The read buffer command brings in the entire buffer following the read header, and the receiving program is responsible for extracting the information it wants based on position. It is intended primarily for diagnostic and other special purposes, and CICS uses it in executing a RECEIVE command only if the BUFFER option is specified. CICS never uses read buffer to read unsolicited terminal input, so the BUFFER option cannot be used on the first RECEIVE of a transaction initiated in this way.

With read modified, the command that CICS normally uses, much less data is transmitted. For the short read keys (CLEAR, CNCL and PAs), only the read header comes in. For other attention keys (ENTER and PFs), the fields on the screen that were changed (those with the MDT on, to be precise) follow the read header. We describe the format in the next section. When transmission occurs because of a trigger field, light pen detect or cursor select, the amount and format of the information is slightly different; these special formats are described in Chapter 42, "Support for special hardware," on page 549. Input from a **program attention** key on an SCS printer is also an exception; see "SCS input" on page 413 for a description of that data stream.

Inbound field format

The next several sections describe the format in which the 3270 transmits data, which you need to understand if you are using terminal control commands. If you are using BMS, you can skip to "Unformatted mode" on page 404, because BMS translates the input for you.

Each modified field comes in as follows:

- SBA order
- Two-byte address of the first *data* position of field
- SF order
- Field contents

Only the non-null characters in the field are transmitted; nulls are skipped, wherever they appear. Thus if an entry does not fill the field, and the field was initially nulls, only the characters keyed are transmitted, reducing the length of the inbound data. Nulls (X'00') are not the same as blanks (X'40'), even though they are indistinguishable on the screen. Blanks get transmitted, and hence you normally initialize fields to nulls rather than to blanks, to minimize transmission.

A 3270 read command can specify that the terminal should return the attribute values along with the field contents, but CICS does not use this option.

Consequently, the buffer address is the location of the first byte of field data, not the preceding attributes byte (as it is in the corresponding outbound data stream).

Note: Special features of the 3270 for input, such as the cursor select key, trigger fields, magnetic slot readers, and so on, produce different input formats. See “Field selection features” on page 551 for details.

Input data stream example

To illustrate an inbound data stream, we assume that an operator using the screen shown in Figure 90 on page 398 did the following:

- Put “123456” in the employee identifier field
- Put “ABC987” in the tag number
- Pressed ENTER, without filling in the state field

Here is the resulting inbound data stream:

Table 29. 3270 input data stream

Bytes	Contents	Notes
1	X'7D'	AID, in this case the ENTER key.
2-3	X'C3C5'	Cursor address: line 3, column 38, where the operator left it after the last data keystroke.
4	X'11'	SBA, indicating that a buffer address follows.
5-6	X'C26E'	Address of line 3, column 15, which is the starting position of the field to follow.
7-12	'123456'	Input, the employee number entered by the operator.
13-15	X'11C3D1'	SBA sequence indicating a buffer address of line 3, column 32.
16	X'1D'	SF, indicating another input field follows.
17-22	'ABC987'	Input field: plate number. Notice that only six characters came in from a field that was eight long, because an operator left the remaining positions null.

Note that the third input field (the state code) does not appear in the input data stream. This is because its MDT did not get turned on; it was set off initially, and the operator did not turn it on by keying into the field. Note also that no SF is required at byte 7 because CICS normally issues a Read Modified All.

Unformatted mode

Even though the high function of the 3270 revolves around its field structure, it is possible to use the 3270 without fields, in what is called **unformatted mode**. In this mode, there are no fields defined, and the entire screen (buffer) behaves as a single string of data, inbound and outbound, much like earlier, simpler terminals.

When you write in unformatted mode, you define no fields in your data, although you can include SBA orders to direct the data to a particular positions on the screen. Data that precedes any SBA order is written starting at the current position of the cursor. (If you use an erase or write command, the cursor is automatically set to zero, at the upper left corner of the screen.)

When you read an unformatted screen, the first three bytes are the read header (the AID and the cursor address), just as when you read a formatted screen. The remaining bytes are the contents of the entire buffer, starting at position zero. There

are no SBA or SF orders present, because there are no fields. If the read command was read modified, the nulls are suppressed, and therefore it is not always possible to determine exactly where on the screen the input data was located.

You cannot use a BMS RECEIVE MAP command to read an unformatted screen. BMS raises the MAPFAIL condition on detecting unformatted input, as explained in “MAPFAIL and other exceptional conditions” on page 504. You can read unformatted data only with a terminal control RECEIVE command in CICS.

Note: The CLEAR key puts the screen into unformatted mode, because it sets the buffer to nulls, thereby erasing all the attributes bytes that demarcate fields.

Chapter 28. CICS support for printing

CICS does not provide special commands for printing, but there are options on BMS and terminal control commands that apply only to printers, and for some printers you use transient data or SPOOL commands. We cover the factors that determine the API and the choices you have in the sections that follow.

There are two issues associated with printing that do not usually occur in other types of end-user communication:

1. There are additional formatting considerations, especially for 3270 printers
2. The task that needs to print may not have direct access to the printer.

In addition, there are two distinct categories of printer, which have different application programming interfaces:

CICS printers

Printers defined as terminals to CICS and managed directly by CICS. They are usually low-speed devices located near the end users, suitable for printing on demand of relatively short documents. The 3289 and 3262 are usually attached as CICS printers.

Non-CICS printers

Printers managed by the operating system or another application. These printers are usually high-speed devices located at the central processing site, appropriate for volume printing that does not have to be available immediately. They may also be advanced function or other printers that require special connections, management, or sharing.

This chapter describes:

- “Formatting for CICS printers”
- “CICS 3270 printers” on page 408
- “CICS 3270 printer options” on page 409
- “Non-3270 CICS printers” on page 412
- “Determining the characteristics of a CICS printer” on page 413
- “Using CICS printers” on page 415
- “Using Non-CICS printers” on page 417
- “Printing display screens” on page 420

Formatting for CICS printers

The application programming interface for writing to a printer terminal is essentially the same as for writing to a display. (This section does not discuss the problem of arranging that your task have the printer as its principal facility; this is discussed in “Using CICS printers” on page 415.)

You can use terminal control commands (SENDS) for any CICS printer, and most of them are supported by BMS too (SEND MAP, SEND TEXT, and SEND CONTROL). “BMS support levels” on page 453 lists the devices that BMS supports. For printers that are components of an outboard controller or LU Type 4, you can use batch data interchange (BDI) commands as well as terminal control and BMS. BDI commands are described in “Using batch data interchange” on page 383.

The choice between using BMS and terminal control is based on the same considerations as it is for a display terminal. Like displays, printers differ widely from one another, both in function and in the implementation of that function, and the differences are reflected in the data streams and device controls they accept.

When you use terminal control commands, your application code must format the output in the manner required by the printer. For line printers and similar devices, formatting has little programming impact. For high-function printers, however, the data stream often is very complex; formatting requires significant application code and introduces device dependencies into program logic.

For some of these terminals, coding effort is greatly reduced by using BMS, which relieves the programmer of creating or even understanding device data streams. BMS also removes most data stream dependencies from the application code so that the same program can support many types of printers, or a mixture of printers and displays, without change. BMS does not remove all device dependencies and imposes a few restrictions on format. It also involves extra path length; the amount depends on how many separate BMS requests you make, the complexity of your requests, and the corresponding path length avoided in your own program.

CICS 3270 printers

Most of the additional format controls for printers that BMS provides are for a specific type of CICS printer, the 3270 printer. A **3270 printer** is any printer that accepts the 3270 data stream—it is the hardcopy equivalent of a 3270 display. It has a page buffer, corresponding to the display buffer of a 3270 display device. (See “The 3270 buffer” on page 390 for an introductory discussion of the 3270 data stream.) We discuss 3270 printers first and defer the simpler, non-3270 printers, until “Non-3270 CICS printers” on page 412.

A 3270 printer accepts two different types of formatting instructions: **buffer control orders** and **print format orders**. Buffer control orders are executed as they are received by the control unit, and they govern the way in which the buffer is filled. These are same orders that are used to format a 3270 display screen. We have already described some of the important ones in “Orders in the data stream” on page 395. For example, SBA (set buffer address) tells the control unit where in the buffer to place the data that follows, SF (start field), which signals an attributes byte and possibly field data, and so on. You can find a complete list in the *IBM 3270 Information Display System Data Stream Programmer's Reference* manual.

In contrast, print format orders are not executed when they are received, but instead are stored in the buffer along with the data. These orders—NL (new line), FF (form feed), and so on—are interpreted only during a print operation, at which time they control the format of the printed output. (They have no effect on displays, other than to occupy a buffer position; they look like blanks on the screen.)

If you are writing to a 3270 printer, you can format with either buffer control orders or print format orders or a mixture of both. We show an example of formatting with buffer control orders in “Outbound data stream sample” on page 398. If you send this same data stream to a 3270 printer, it prints an image of the screen shown in Figure 90 on page 398. You might choose to format printed output with buffer control orders so that you can send the same data stream to a display and a printer.

On the other hand, you might choose to format with print format orders so that you can send the same stream to a 3270 printer and a non-3270 printer (print format orders are the same as the format controls on many non-3270 printers). See the discussion of the NLEOM option on page 410 for more details about this choice.

Here is a data stream using print format orders that produces the same *printed* output as the data stream on page 398, which uses buffer control orders.

Table 30. Example of data stream using print control orders

Bytes	Contents	Notes
1	X'FF'	“Formfeed” (FF) order, to cause printer to space to a new page.
2-23	blanks	22 blanks to occupy columns 1-22 on first line.
24-33	Car Record	Text to be printed, which appears in the next available columns (23-32) on line 1.
34	X'1515'	Two successive “new line” (NL) orders, to position printer to beginning of third line.
35-80	Employee No: _____ Tag _____ State: __	Text to be printed, starting at first position of line 3.
81	X'19'	“End-of-message” (EM) print order, which stops the printing.

Notice that the field structure is lost when you use print format orders. This does not matter ordinarily, because you do not use the printer for input. However, even if you format with print control orders, you might need to use buffer control orders as well, to assign attributes like color or underscoring to an area of text.

CICS 3270 printer options

For BMS, the special controls that apply to 3270 printers take the form of command options:

- PRINT
- ERASE
- L40, L64, L80 and HONEOM
- NLEOM
- FORMFEED
- PRINTERCOMP

In terminal control commands, ERASE is also expressed as an option, but the other controls are expressed directly in the data stream. The *IBM CICS/OS/VS 3270 Data Stream Device Guide* and the *IBM 3270 Information Display System Data Stream Programmer's Reference* tell you how to encode them; the discussion that follows explains what they do.

PRINT option and print control bit

Writing to a 3270 display or printer updates the device buffer. On a display, the results are reflected immediately on the screen, which is driven from the buffer. For a printer, however, there might be no visible effect, because printing does not occur until you turn on the appropriate bit in the “write control character”. (The WCC is part of the 3270 data stream; see “Write control character” on page 391.) For BMS, you turn on the print bit by specifying the PRINT option on a SEND MAP, SEND TEXT, or SEND CONTROL command, or in the map used with SEND MAP. If you are using terminal control SENDs, you must turn on the print bit with the CTLCHAR option.

A terminal write occurs on every terminal control SEND, and on every SEND MAP, SEND TEXT, or SEND CONTROL unless you are using the ACCUM or PAGING options. ACCUM delays writing until a page is full or the logical message is ended. When you use ACCUM, you should use the same print options on every SEND

command for the same page. PAGING defers the terminal writes to another task, but they are generated in the same way as without PAGING.

The fact that printing does not occur until the print bit is on allows you to build the print buffer in stages with multiple writes and to change data or attribute bytes already in the buffer. That is, you can use the hardware to achieve some of the effects that you get with the ACCUM option of BMS. The NLEOM option affects this ability, however; see the discussion below.

ERASE option

Like the 3270 display buffer, the 3270 printer buffer is cleared only when you use a write command that erases. You do this by specifying the ERASE option, both for BMS and terminal control SENDs. If the printer has the alternate screen size feature, the buffer size is set at the time of the erase, as it is for a display. Consequently, the first terminal write in a transaction should include erasing, to set the buffer to the size required for the transaction and to clear any buffer contents left over from a previous transaction.

Line width options: L40, L64, L80, and HONEOM

In addition to the print bit, the write control character contains a pair of bits that govern line length on printing. If you are using terminal control commands, you use the CTLCHAR option to set these bits. For BMS, the default is the one produced by the HONEOM option, which stands for “honor end-of-message”. With this setting, the printer formats according to the buffer control and print format orders only, stopping printing at the first EM (end-of-message) character in the buffer. Only if you attempt to print beyond the maximum width for the device (the platen width) does the printer move to a new line on its own.

However, you also can specify that the line length is fixed at 40, 64, or 80 characters (the L40, L64 and L80 options, respectively). If you do, the printer ignores certain print format orders, moves to a new line when it reaches the specified line size, and prints the entire buffer. The print format orders that are ignored are NL (new line), CR (carriage return), and EM (end-of-message). Instead they are simply printed, as graphics.

If you use L40, L64, or L80 under BMS, you should use only the value that corresponds to the page width in your terminal definition (see “Determining the characteristics of a CICS printer” on page 413). The reason is that BMS calculates buffer addresses based on the page size, and these addresses are wrong if you use a different page width.

NLEOM option

BMS ordinarily uses buffer control orders, rather than print format orders, to format for a 3270 printer, whether you are using SEND TEXT or SEND MAP. However, you can tell BMS to use print format orders only, by specifying the NLEOM option. If you do, BMS formats the data entirely with blanks and NL (new line) characters, and inserts an EM (end-of-message) character after your data. NLEOM implies HONEOM. (NLEOM support requires **standard** BMS; it is not available in minimum BMS.)

You might want to do this in order to maintain compatibility with an SCS printer (print format orders are compatible with the corresponding SCS control characters). The following operational differences might cause you to choose or avoid NLEOM.

Blank lines

The 3270 printer suppresses null lines during printing. That is, a line that has no data fields and appears blank on the display screen is omitted when the same map is sent to a printer. Under BMS, you can force the printed form to look exactly like the displayed form by placing at least one field on every line of the screen; use a field containing a single blank for lines that would otherwise be empty. Specifying NLEOM also has this effect, because BMS uses a new line character for every line, whether or not there is any data on it.

Multiple sends

With NLEOM, data from successive writes is simply stacked in the buffer, since it does not contain positioning information. However, BMS adds an EM (end-of-message) character at the end of data on each SEND with NLEOM, unless you are using the ACCUM option. When printing occurs, the first EM character stops the printing, so that only the data from the first SEND with NLEOM (and any unerased data up to that point in the buffer) gets printed. The net effect is that you cannot print a buffer filled with multiple SEND commands with NLEOM unless you use the ACCUM option.

Page width

BMS always builds a page of output at a time, using an internal buffer whose size is the number of character positions on the page. (See “Determining the characteristics of a CICS printer” on page 413 for a discussion of how BMS determines the page size.) If you are using buffer control orders to format, the terminal definition must specify a page width of 40, 64, 80 or the maximum for the device (the platen size); otherwise your output might not be formatted correctly. If you are using NLEOM, on the other hand, the terminal definition may specify any page width, up to the platen size.

Total page size

If you are using buffer control orders, the product of the number of lines and the page width must not exceed the buffer size, because the buffer is used as an image of the page. Unused positions to the right on each line are represented by null characters. If you use NLEOM, however, BMS does not restrict page size to the buffer capacity. BMS builds the page according to the page size defined for the terminal and then compresses the stream using new-line characters where possible. If the resulting stream exceeds the buffer capacity, BMS uses multiple writes to the terminal to send it.

FORMFEED

The FORMFEED option causes BMS to put a form feed print format order (X'0C') at the beginning of the buffer, provided that the printer is defined as capable of advancing to the top of the form (with the FORMFEED option in the associated TYPETERM definition). CICS ignores a form feed request for a printer defined without this feature.

If you issue a SEND MAP using a map that uses position (1,1) of the screen, you overwrite the order and lose the form feed. This occurs whether you are using NLEOM or not.

If you use FORMFEED and ERASE together on a SEND CONTROL command, the results depend on whether NLEOM is present. Without NLEOM, SEND CONTROL FORMFEED ERASE sends the form feed character followed by an entire page of null lines. The printer suppresses these null lines, replacing them with a single blank line. With NLEOM, the same command sends the form feed character

followed by one new line character for each line on the page, so that the effect is a full blank page, just as it is on a non-3270 printer.

PRINTERCOMP option

When you SEND TEXT to a printer, there is one additional option that affects page size. This is the PRINTERCOMP option, which is specified in the PROFILE associated with the transaction you are executing, rather than on individual SEND TEXT commands. (In the default profile that CICS provides, the PRINTERCOMP value is NO.)

Under PRINTERCOMP(NO), BMS produces printed output consistent with what it would send to a 3270 display. For the display, BMS precedes the text from each SEND TEXT command with an attribute byte, and it also starts each line with an attribute byte. These attribute bytes take space on the screen, and therefore BMS replaces them with blanks for printers if PRINTERCOMP is NO. If PRINTERCOMP is YES, BMS suppresses these blanks, allowing you to use the full width of the printer and every position of the buffer. New line characters that you embed in the text are still honored with PRINTERCOMP(YES), as they are with PRINTERCOMP(NO).

You should use PRINTERCOMP(NO) if you can, for compatibility with display devices and to ensure consistent results if the application uses different printer types, even though it reduces the usable line width by one position.

Non-3270 CICS printers

A **non-3270 printer** is any printer that does not accept the 3270 data stream, such as an SNA character set (SCS) printer. The terminology is somewhat confusing, because a non-3270 printer can be a 3270-family device, and many devices, like the 3287 and 3262, can be either 3270 printers or SCS (non-3270) printers, depending on how they are defined at the control unit.

There are special considerations for non-3270 printers, although not so many as for 3270 printers. Non-3270 printers do not have page buffers, and therefore do not understand buffer control orders. Formatting is accomplished entirely with print control orders. For compatibility with 3270 printers, BMS formats for them by constructing an image of a page in memory, and always prints a full page at a time. However, you can define any size page, provided you do not exceed the platen width, as there is no hardware buffer involved. BMS transmits as many times as required to print the page, just as it does for a 3270 printer using the NLEOM option.

BMS formats for these printers with blanks and NL (new line) characters. It uses form feed (FF) characters as well if the definition of your terminal indicates form feed support.

BMS also uses horizontal tabs to format if the terminal definition has the HORIZFORM option and the map contains HTAB specifications. Similarly, it uses vertical tabs if the terminal definition specifies VERTICALFORM and your map includes VTAB. Tab characters can shorten the data stream considerably. If tabs are used, BMS assumes that the current task, or some earlier one, has already set the tabs on the printer. On an SCS printer, you set the tabs with a terminal control SEND command, as explained in the *IBM CICS/OS/VS 3270 Data Stream Device Guide*. For other non-3270 printers, you should consult the appropriate device guide.

For SEND TEXT to an SCS printer, BMS does not recognize any non-3270 control codes in the input datastream except newline (X'15') and set attribute (X'28'). All other characters are assumed to be display characters. In particular, the datastream might be affected if you attempt to use the transparency control order (X'35') under BMS. This control order normally causes the data that follows it to be ignored (the next byte contains the length of the data to be ignored). However, because BMS does not recognize the X'35' control order, it processes the data that follows the transparency control order as if it were a normal part of the datastream. If this data cannot be processed correctly, BMS might remove it from the datastream. For example, if the X'28' character is encountered in the transparency sequence it will be mistaken for a set attribute control order, in which case the two bytes following it will be mistaken for an attribute description, and all three bytes might be removed from the datastream. The X'0C' character (formfeed) is also liable to be removed from the datastream. If you want to send a datastream including a transparency sequence which contains characters that may be recognized and altered by BMS, the recommended method is to use a terminal control SEND command, rather than BMS.

SCS input

SCS printers also have limited *input* capability, in the form of “program attention” keys. These keys are *not* like the PA keys described in “Attention keys” on page 401, however. Instead they transmit an unformatted data stream consisting of the characters ‘APAK nn’, where “nn” is the 2-digit PA key number—‘APAK 01’ for PA key 1, for example.

You can capture such input by defining a transaction named ‘APAK’ (APAK is the transaction identifier, not the TASKREQ attribute value, because SCS inputs do not look like other PA key inputs.) A program invoked by this transaction can determine which PA key was pressed by issuing a RECEIVE and numeric positions of the input.

Determining the characteristics of a CICS printer

If you are writing a program that supports more than one type of CICS printer, you may need to determine the characteristics of a particular printer. As we explained in connection with terminals generally, you can use the ASSIGN and INQUIRE TERMINAL commands for this purpose. Table 25 on page 376 lists the ASSIGN options that apply to terminals, including several that are specific to printers.

The INQUIRE TERMINAL options that apply specifically to printers and the corresponding parameters in the terminal definition are shown in Table 31:

Table 31. INQUIRE TERMINAL options for printers

INQUIRE option	Source in TERMINAL or TYPETERM definition	Description
PAGEHT	x of PAGESIZE(x,y)	Number of lines per page (for alternate screen size terminals, reflects default size)
PAGEWD	y of PAGESIZE(x,y)	Number of characters per line (for alternate screen size terminals, reflects default size)
DEFPAGEHT	x of PAGESIZE(x,y)	Number of lines per page in default mode (alternate screen size terminals only)
DEFPAGEWD	y of PAGESIZE(x,y)	Number of characters per line in default mode (alternate screen size terminals only)

Table 31. INQUIRE TERMINAL options for printers (continued)

INQUIRE option	Source in TERMINAL or TYPETERM definition	Description
ALTPAGEHT	x of ALTPAGE(x,y)	Number of lines per page in alternate mode (alternate screen size terminals only)
ALTPAGEWD	y of ALTPAGE(x,y)	Number of characters per line in alternate mode (alternate screen size terminals only)
DEVICE	DEVICE	The device type (see the <i>CICS System Programming Reference</i> for possible values)
TERMMODEL	TERMMODEL	The model number of the terminal (either 1 or 2)

BMS page size, 3270 printers

BMS uses both the terminal definition and the profile of the transaction that is running to determine the page size of a CICS printer. The profile is used when the terminal has the alternate screen size feature, to determine whether to use default or alternate size. (The default profile in CICS specifies “default” size for the screen.) Table 32 lists the values used.

Table 32. Priority of parameters defining BMS page size. BMS uses the first value in the appropriate column that has been specified in the terminal definition.

Terminals with alternate screen size, using alternate size	Terminals with alternate screen size, using default size	Terminals without alternate screen size feature
ALTPAGE	PAGESIZE	PAGESIZE
ALTSCREEN	DEFSCREEN	TERMMODEL
DEFSCREEN	TERMMODEL	(12,80)
TERMMODEL	(12,80)	
(12,80)		

The definition of a “page” is unique to BMS. If you are printing with terminal control SEND commands, you define what constitutes a page, within the physical limits of the device, by your print format. If you need to know the buffer size to determine how much data you can send at once, you can determine this from the SCRNH and SCRND values returned by the ASSIGN command.

Supporting multiple printer types

When you are writing programs to support printers that have different page sizes, it is not always possible to keep device dependencies like page size out of the program. However, BMS helps with this problem in two ways.

1. You can refer to a map generically and have BMS select the map that was designed for the terminal associated with your task (see the discussion of map suffixes in “Device-dependent maps” on page 472).
2. If you are using SEND TEXT, BMS breaks the text into lines at word boundaries, based on the page size of the receiving terminal. You can also request header and trailer text on each page.

Using CICS printers

As we noted at the start of the chapter, the second issue that frequently arises in printing concerns ownership of the printer. Requests for printing often originate from a user at a display terminal. The task that processes the request and generates the printed output is associated with the user's terminal and therefore cannot send output directly to the printer.

If your task does not own the printer it wants to use, it must create another task, which does, to do the work. These are the ways to do this:

1. Create the task with a START command.
2. Write to an intrapartition transient data queue that triggers the task.
3. Direct the output to the printer in a BMS ROUTE command.
4. Use the ISSUE PRINT command, if you need only a screen copy.

Printing with a START command

The first technique for creating the print task is to issue a START command in the task that wants to print. The command names the printer as the terminal required by the STARTed task in the TERMID option and passes the data to be printed, or instructions on where to find it, in the FROM option. START causes CICS to create a task whose principal facility is the designated terminal when that terminal is available.

The program executed by the STARTed task, which you must supply, retrieves the data to be printed (using a RETRIEVE command), and then writes it to its terminal (the printer) with SEND, SEND MAP, or SEND TEXT. For example:

```
      :  
(build output in OUTAREA, formatted as expected by the STARTed task)  
      EXEC CICS START TRANSID(PRNT) FROM(OUTAREA) TERMID(PRT1)  
              LENGTH(OUTLNG) END-EXEC.  
      :
```

Figure 91. Task that wants to print (on printer PRT1)

```
      :  
      EXEC CICS RETRIEVE INTO(INAREA) LENGTH(INLNG) END-EXEC.  
      :  
(do any further data retrieval and any formatting required)  
      EXEC CICS SEND TEXT FROM(INAREA) LENGTH(INLNG) ERASE PRINT END-EXEC.  
      :  
(repeat from the RETRIEVE statement until a NODATA condition arises)
```

Figure 92. STARTed task (executing transaction PRNT)

The task associated with the printer loops until it exhausts all the data sent to it, in case another task sends data to the same printer before the current printing is done. Doing this saves CICS the overhead of creating new tasks for outputs that arrive while earlier ones are still being printed; it does not change what finally gets printed, as CICS creates new tasks for the printer as long as there are unprocessed START requests.

Printing with transient data

The second method for creating the print task involves transient data. A CICS intrapartition transient data queue can be defined to have a property called a "trigger". When the number of items on a queue with a trigger reaches the trigger

value, CICS creates a transaction to process the queue. The queue definition tells CICS what transaction this task executes and what terminal, if any, it requires as its principal facility.

You can use this mechanism to get print data from the task that generates it to a task that owns the printer. A transient data queue is defined for each printer where you direct output in this way. A task that wants to print puts its output on the queue associated with the required printer (using WRITEQ TD commands). When enough items are on the queue and the printer is available, CICS creates a task to process the queue. (For this purpose, the trigger level of “enough” is usually defined as just one item.) The triggered task retrieves the output from the queue (with READQ TD commands) and writes it to its principal facility (the printer), with SEND, SEND MAP, or SEND TEXT commands.

As in the case of a STARTed printer task, you have to provide the program executed by the task that gets triggered. The sample programs distributed with CICS contain a complete example of such a program, called the “order queue print sample program”. The *CICS 4.1 Sample Applications Guide* describes this program in detail, but the essentials are as follows:

Task that wants to print (on printer PRT1):

```
⋮
(do any formatting or other processing required)
EXEC CICS WRITEQ TD QUEUE('PRT1') FROM(OUTAREA)
      LENGTH(OUTLNG) END-EXEC.
⋮
```

Task that gets triggered:

```
⋮
EXEC CICS ASSIGN QNAME(QID) END-EXEC.
EXEC CICS READQ TD QUEUE(QID) INTO(INAREA) LENGTH(INLNG)
      RESP(RESPONSE) END-EXEC.
IF RESPONSE = DFHRESP(QZERO) GO TO END-TASK.
⋮
(do any error checking, further data retrieval and formatting required)
EXEC CICS SEND FROM(INAREA) LENGTH(INLNG) END-EXEC.
⋮
(repeat from READQ command)
```

The print task determines the name of its queue using an ASSIGN command rather than a hard-coded value so that the same code works for any queue (printer).

Like its START counterpart, this task loops through its read and send sequence until it detects the QZERO condition, indicating that the queue is empty. While this is just an efficiency issue with the STARTed task, it is critical for transient data; otherwise unprocessed queue items can accumulate under certain conditions. (See “Automatic transaction initiation (ATI)” on page 595 for details on the creation of tasks to process transient data queues with triggers.)

If you use this technique, you need to be sure that output to be printed as a single unit appears either as a single item or as consecutive items on the queue. There is no fixed relationship between queue items and printed outputs; packaging arrangements are strictly between the programs writing the queue and the one reading it. However, if a task writes multiple items that need to be printed together, it must ensure that no other task writes to the queue before it finishes. Otherwise the printed outputs from several tasks may be interleaved.

If the TD queue is defined as recoverable, CICS prevents interleaving. Once a task writes to a recoverable queue, CICS delays any other task that wants to write until the first one commits or removes what it has written (by SYNCPOINT or end of task). If the queue is not recoverable, you need to perform this function yourself. One way is to ENQUEUE before writing the first queue item and DEQUEUE after the last. (See Chapter 47, “Transient data control,” on page 593 for a discussion of transient data queues.)

Printing with BMS routing

A task also can get output to a printer other than its principal facility with BMS routing. This technique applies only to BMS logical messages (the ACCUM or PAGING options) and thus is most appropriate when you are already building a logical message.

When you complete a routed message, CICS creates a task for each terminal named in a route list. This task has the terminal as its principal facility, and uses CSPG, the CICS-supplied transaction for displaying pages, to deliver the output to the printer. So routing is similar in effect to using START commands, but CICS provides the program that does the printing. (See Chapter 39, “Message routing,” on page 527 for more information about routing.)

Using Non-CICS printers

Here are the steps to use a printer managed outside CICS:

1. Format your output in the manner required by the application or subsystem that controls the printer you wish to use.
2. Deliver the output to the application or subsystem that controls the printer in the form required by that application.
3. If necessary, notify that application that the output is ready for printing.

Formatting for non-CICS printers

For some printers managed outside CICS, you can format output with BMS, as we explain in “Programming for non-CICS printers” on page 418. However, for most printers, you need to meet the format requirements of the application that drives the printer. This may be the device format or an intermediate form dictated by the application. For conventional line printers, formatting is simply a matter of producing line images and, sometimes, adding carriage-control characters.

Non-CICS printers: Delivering the data

Print data is usually conveyed to an application outside of CICS by placing the data in an intermediate file, accessible to both CICS and the application. The type of file, as well as the format within the file, is dictated by the receiving application. It is usually one of those listed in the first column of Table 33. The second column of the table shows which groups of CICS commands you can use to create such data.

Table 33. Intermediate files for transferring print data to non-CICS printers

File type	Methods for writing the data
Spool files	CICS spool commands (SPOOLOPEN, SPOOLWRITE, etc.) Transient data commands (WRITEQ TD) Terminal control and BMS commands (SEND, SEND MAP, etc.)
BSAM	CICS spool commands (SPOOLOPEN, SPOOLWRITE, etc.) Transient data commands (WRITEQ TD)
VSAM	CICS file control commands (WRITE)

Table 33. Intermediate files for transferring print data to non-CICS printers (continued)

File type	Methods for writing the data
DB2	EXEC SQL commands
IMS	EXEC DLI commands or CALL DLI statements

Programming for non-CICS printers

If you are using VSAM, DB2, or IMS, the CICS application programming commands you can use are determined by the type of file you are using.

For BSAM and spool files, however, you have a choice. The CICS definition of the file (or its absence) determines which commands you use. The file may be:

- An extra-partition transient data queue (see Chapter 47, “Transient data control,” on page 593 for information on transient data queues)
- The output half of a sequential terminal (see “Using sequential terminal support” on page 381 and “Support for non-3270 terminals” on page 470)
- A spool file (see Chapter 29, “CICS interface to JES,” on page 423)

Both transient data queue definitions and sequential terminal definitions point to an associated data definition (DD) statement in the CICS start-up JCL, and it is this DD statement that determines whether the file is a BSAM file or a spool file. Files created by CICS spool commands do not require definition before use and are spool files by definition.

If the printing application accepts BSAM or spool file input, there are several factors to consider in deciding how to define your file to CICS:

System definitions

Files created by the SPOOLOPEN command do not have to be defined to CICS or the operating system, whereas transient data queues and sequential terminals must be defined to both before use.

Sharing among tasks

A file defined as a transient data queue is shared among all tasks. This allows you to create a print file in multiple tasks, but it also means that if your task writes multiple records to the queue that must be printed together (lines of print for a single report, for example), you must include enqueue logic to prevent other tasks from writing their records between yours. This is the same requirement that was cited for intrapartition queues in “Printing with transient data” on page 415. In the case of extra-partition transient data, however, CICS does not offer the recoverability solution, and your program must prevent the interspersing itself.

In contrast, a file created by a SPOOLOPEN can be written only by the task that created it. This eliminates the danger of interleaving output, but also prevents sharing the file among tasks.

A spool file associated with a sequential terminal can be written by only one task at a time (the task that has the terminal as its principal facility). This also prevents interleaving, but allows tasks to share the file serially.

Release for printing

Both BSAM and spool files must be closed in order for the operating system to pass them from CICS to the receiving application, and therefore printing does not begin until the associated file is closed. Files created by SPOOLOPEN are closed automatically at task end, unless they have

already been closed with a SPOOLCLOSE command. In contrast, an extrapartition transient data queue remains open until some task closes it explicitly, with a SET command. (It must be reopened with another SET if it is to be used subsequently.) So transient data gives you more control over release of the file for processing, at the cost of additional programming.

A file that represents the output of a sequential terminal does not get closed automatically (and so does not get released for printing) until CICS shutdown, and CICS does not provide facilities to close it earlier. If you use a sequential terminal to pass data to a printer controlled outside of CICS, as you might do in order to use BMS, you should be aware of this limitation.

Formatting

If you define your file as a sequential terminal, you can use BMS to format your output. This feature allows you to use the same maps for printers managed outside of CICS—for example, line printers managed by the MVS job entry subsystem (JES)—that you use for CICS display and printer terminals.

If you choose this option, remember that BMS always sends a page of output at a time, using the page size in the terminal definition, and that the data set representing the output from a sequential terminal is not released until CICS shutdown.

Spool file limits

Operating systems identify spool files by assigning a sequential number. There is an upper limit to this number, after which numbers are reused. The limit is usually very large, but it is possible for a job that runs a very long time (as CICS can) and creates a huge number of spool files (as an application under CICS can) to exceed the limit. If you are writing an application that generates a very large number of spool files, consult your systems programmer to ensure that you are within system limits. A new spool file is created at each SPOOLOPEN statement and each open of a transient data queue defined as a spool file.

Notifying the print application

When you deliver the data to a print application outside CICS, you might need to notify the application that you have data ready to process. You do not need to do this if the application runs automatically and knows to look for your data. For example, to print on a printer owned by the MVS job entry system (JES), all you need to do is create a spool file with the proper routing information. JES does the rest.

However, sometimes you need to submit a job to do the processing, or otherwise signal an executing application that you have work for it.

To submit a batch job from a CICS task, you need to create a spool file which contains the JCL for the job, and you need to direct this file to the JES internal reader. You can create the file in any of the three ways listed for spool files in Table 33 on page 417, although if you use a sequential terminal, the job does not execute until CICS shuts down, as noted earlier. For files written with spool commands, the information that routes the file to the JES internal reader is specified in the SPOOLOPEN command. For transient data queues and sequential terminals, the routing information appears on the first record in the file, the “JOB card”.

The output to be printed can be embedded in the batch job (as its input) or it can be passed separately through any form of data storage that the job accepts.

Printing display screens

If your printing requirement is simply to copy a display screen to a printer, you have choices additional to those already described. Some of these are provided by the terminal hardware itself, and some by CICS. Some of the CICS support also depends on hardware features, and so your options depend on the type of terminals involved and, in some cases, the way in which they are defined to CICS. See the *CICS Resource Definition Guide* for more detail on copying.

CICS print key

The first such option is the **CICS print key** (also called the **local copy key**). This allows a user to request a printed copy of a screen by pressing a program attention key, provided the terminal is a 3270 display or a display in 3270 compatibility mode. Print key support is optional in CICS; the system programmer decides whether to include it and what key is assigned. The default is PA1. (See the PRINT option in the *CICS System Definition Guide*.)

The print key copies the display screen to the first available printer among those defined as eligible. Which printers are eligible depends on the definition of the display terminal from which the request originates, as follows:

- For VTAM 3270 displays defined without the “printer-adapter” feature, the printers named in the PRINTER and ALTPRINTER options of the terminal definition are eligible. PRINTER is be used if available; ALTPRINTER is second choice. If both are unavailable, the request is queued for execution when PRINTER becomes available.
- For the 3270 compatibility mode of the 3790 and a 3650 host conversational (3270) logical unit, the same choices apply.
- For VTAM 3270 displays defined with the printer-adapter feature, copying is limited to printers on the same control unit as the display. The printer authorization matrix within the control unit determines printer eligibility.
- For a 3270 compatibility mode logical unit of the 3790 with the printer-adapter feature, the 3790 determines eligibility and allocates a printer for the copy.
- For a 3275 with the printer-adapter feature, the print key prints the data currently in the 3275 display buffer on the 3284 attached to the display.

Where CICS chooses the printer explicitly, as it does in the first three cases above, the printer has to be in service and not attached to a task to be “available” for a CICS print key request. Where a control unit or subsystem makes the assignment, availability and status are determined by the subsystem. The bracket state of the device usually determines whether it is available or not.

ISSUE PRINT and ISSUE COPY

An application can initiate copying a screen to a printer as well as the user, with the ISSUE PRINT and ISSUE COPY commands. ISSUE PRINT simulates the user pressing the CICS print key, and printer eligibility and availability are the same as for CICS print key requests.

There is also a command you can use to copy a screen in a task that owns the printer, as opposed to the task that owns the terminal which is to be copied. This is the ISSUE COPY command. It copies the buffer of the terminal named in the TERMDID option to the buffer of the principal facility of the issuing task. The method

of copying and the initiation of printing once the copy has occurred is controlled by the “copy control character” defined in the CTLCHAR option of the ISSUE COPY command; see the *IBM CICS/OS/VS 3270 Data Stream Device Guide* for the bit settings in this control character. The terminal whose buffer is copied and the printer must both be 3270 logical units, and they must be on the same control unit.

Hardware print key

Some 3270 terminals also have a **hardware print key**. Pressing this key copies the screen to the first available and eligible printer on the same control unit as the display. This function is performed entirely by the control unit, whose configuration and terminal status information determine eligibility and availability. If no printer is available, the request fails; the user is notified by a symbol in the lower left corner of the screen and must retry the request later.

BMS screen copy

Both the CICS and hardware print keys limit screen copies to a predefined set of eligible printers, and if more than one printer is eligible, the choice depends on printer use by other tasks. For screens created as part of a BMS logical message, a more general screen copy facility is available. Users can print any such screen with the “page copy” option of the CICS-supplied transaction for displaying logical messages, CSPG. With page copy, you name the specific printer to receive the output, and it does not have to be on the same control unit as the display. CSPG is described in the *CICS Supplied Transactions* manual.

Chapter 29. CICS interface to JES

CICS provides a programming interface to **JES** (the Job Entry Subsystem component of MVS) that allows CICS applications to create and retrieve **spool** files. Spool files are managed by JES and are used to buffer output directed to low-speed **peripheral** devices (printers, punches, and plotters) between the job that creates them and actual processing by the device. Input files from card readers are also spool files and serve as buffers between the device and the jobs that use the data.

The interface consists of five commands:

- SPOOLOPEN INPUT, which opens a file for input
- SPOOLOPEN OUTPUT, which opens a file for output
- SPOOLREAD, which retrieves the next record from an input file
- SPOOLWRITE, which adds one record to an output file
- SPOOLCLOSE, which closes the file and releases it for subsequent processing by JES

“Input” and “output” here refer to the CICS point of view here; what is spool output to one job is always spool input to another job or JES program.

These commands can be used with either the JES2 or JES3 form of JES, although some restrictions apply to each (see “Spool interface restrictions” on page 425). The term JES refers to both. You can support the requirements of other products to exchange files with other systems connected through a JES remote spooling communications subsystem (RSCS) network.

You can use the spool commands to do the following types of things:

- Create an (output) file for printing or other processing by JES. JES manages most of the “unit record” facilities of the operating system, including high-speed printers, and card readers. In order to use these facilities, you pass the data to be processed to JES in a spool file. See Figure 93

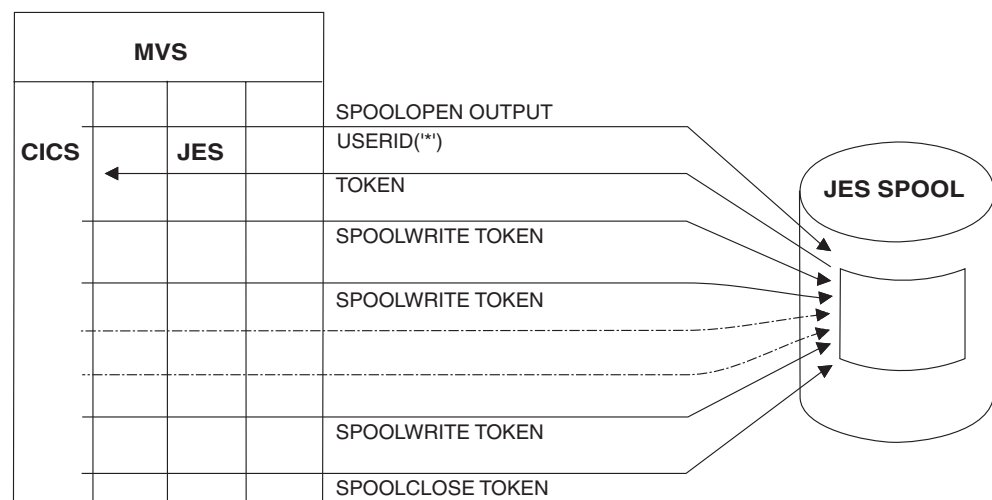


Figure 93. Create a file and write it to the JES spool

- Submit a batch job to MVS. Spool files directed to the JES “internal reader” are treated as complete jobs and executed.
- Create an (output) file to pass data to another job (outside of your CICS), that runs under MVS.
- Retrieve data passed from such a job. See Figure 94

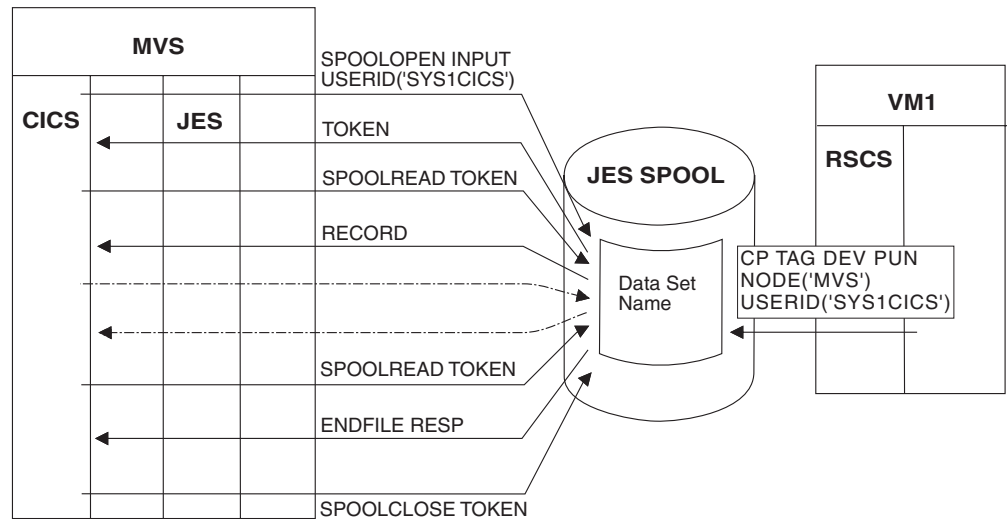


Figure 94. Retrieve data from the JES spool

- Create a file to pass data to another operating system, such as VM, VSE/ESA, or an MVS system other than the one under which your CICS is executing. See Figure 95.

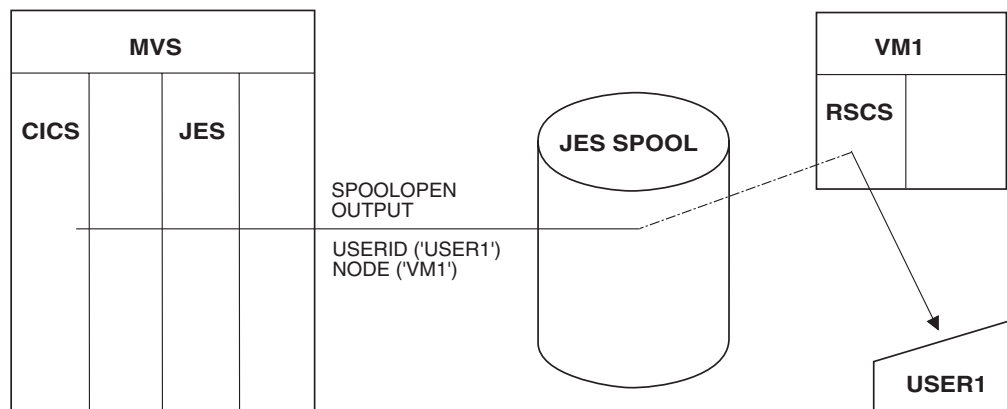


Figure 95. Send a written file to a remote destination

This chapter describes:

- “Using the CICS interface to JES” on page 425
- “Creating output spool files” on page 425
- “Reading input spool files” on page 426
- “Identifying spool files” on page 427
- “Examples of SPOOL commands” on page 430

Using the CICS interface to JES

To use the CICS interface to JES, you must define the DFHSIT SPOOL=YES system initialization parameter in your CICS startup JCL.

You must specify RESP or NOHANDLE on the EXEC CICS SPOOLCLOSE, SPOOLOPEN, SPOOLREAD, and SPOOLWRITE commands. RESP bears a one-to-one correspondence with HANDLE CONDITION. If you do not code RESP, your program abends. You can also code the RESP2 option.

Transactions that process SYSOUT data sets larger than 1000 records, either for INPUT or for OUTPUT, are likely to have a performance impact on the rest of CICS. When you cannot avoid such a transaction, you should carefully evaluate general system performance. You should introduce a pacing mechanism if the effects on the rest of CICS are unacceptable.

All access to a JES spool file must be completed within one logical unit of work. Issuing an EXEC CICS SYNCPOINT command implicitly issues a SPOOLCLOSE command for any open report.

Spool interface restrictions

There are internal limits in JES that you should consider when you are designing applications. Some apply to JES2, some to JES3 and some to both. In particular:

- JES2 imposes an upper limit on the total number of spool files that a single job (such as CICS) can create. If CICS exceeds this limit during its execution, subsequent SPOOLOPEN OUTPUT commands fail with the ALLOCERR condition.
- JES3 does not impose such a limit explicitly, but for both JES2 and JES3, some control information for each file created persists for the entire execution of CICS. For this reason, creating very large numbers of spool files can stress JES resources; you should consult your system programmer before designing such an application.
- Spool files require other resources (buffers, queue elements, disk space) until they are processed. You need to consult your systems staff if you are producing very large files or files that may wait a long time for processing at their destinations.
- Code NODE('*') and USERID('*') if you want to specify the local spool file and to enable the OUTDESCR operand to override the NODE and USERID operands. Do not use NODE('*') with any other userid. If the NODE and USERID operands specify explicit identifiers, the OUTDESCR operands cannot override them.
- Ensure that your system is defined so that data sets produced by CICS are not in HELD status in JES. CICS does not search for data sets in HELD status when the EXEC CICS SPOOLOPEN INPUT command is issued.

Creating output spool files

To create an output spool file, your program starts by issuing a SPOOLOPEN OUTPUT command, to allocate an output data set.. The NODE and USERID options on the command tell JES what to do with the file when it is complete, and there are other options to convey formatting and other processing to JES if appropriate. SPOOLOPEN returns a unique token in the TOKEN field, which must be used in all subsequent SPOOLWRITE and SPOOLCLOSE commands to identify the file being written.

Thereafter, the task puts data into the file with SPOOLWRITE commands that specify the token value that was returned on the SPOOLOPEN OUTPUT command. Spool files are sequential; each SPOOLWRITE adds one record to the file. When the file is complete, the task releases the file to JES for delivery or processing by issuing a SPOOLCLOSE with the token that identifies the file.

A task can create multiple output spool files, and it can have more than one open at a time; operations on different files are kept separate by the token. However, a spool file cannot be shared among tasks, or across logical units of work in the same task. It can be written only by the task that opened it, and if that task fails to close the file before a SYNCPOINT command or task end, CICS closes it automatically at these points.

If the node is a remote system, the data set is queued on the JES spool against the destination userid. The ID of this destination user was specified on the SPOOLOPEN OUTPUT USERID parameter. If the node is a remote VM system, the data is queued in the VM RDR queue for the ID that was specified on the same USERID parameter.

Note: If you want the job you submit to execute as soon as possible, you should end your spool file with a record that contains /*EOF in the first five characters. This statement causes JES to release your file for processing, rather than waiting for other records to fill the current buffer before release.

Using the MVS internal reader

You can use the USERID parameter to specify that your output is to be written to the MVS internal reader. To use CICS SPOOL commands for this purpose, specify USERID("INTRDR") and also use an explicit node name. Do not use NODE('*'). INTRDR is an IBM-reserved name identifying the internal reader. If you specify USERID("INTRDR"), the output records written by your SPOOLWRITE commands must be JCL statements, starting with a JOB statement. Also ensure that you specify the NOCC option on the SPOOLOPEN command. The system places your output records for the internal reader into a buffer in your address space. When this buffer is full, JES places the contents on the spool; later, JES retrieves the job from the spool. (See "Identifying spool files" on page 427 for more information about the naming of spool files.

Reading input spool files

The command sequence for reading a spool file is similar to that for creating one. You start with a SPOOLOPEN INPUT command that selects the file. Then you retrieve each record with a SPOOLREAD command. When the file is exhausted or you have read as much as required, you end processing with a SPOOLCLOSE command. CICS provides you with a token to identify the particular file when you open it, just as it does when you open an output file, and you use the token on all subsequent commands against the file.

Similar to an output spool file, an input spool file is exclusive to the task that opened it. No other task can use it until the first one closes it. The file must be read in the same logical unit of work that opened it, and CICS closes it automatically at a SYNCPOINT command or at task end if the task does not do so. However, you can close the file in such a way that your task (or another one) can read it again from the beginning.

In contrast to output files, a task can have only one spool file open for input at once. Moreover, *only one* CICS task can have a file open for input at any given time. This single-threading of input spool files has several programming implications:

- A task reading a spool file should keep it open for as little time as possible, and should close it explicitly, rather than letting CICS do so as part of end-of-task processing. You might want to transfer the file to another form of storage if your processing of individual records is long.
- If another task is reading a spool file, your SPOOLOPEN INPUT command fails with a SPOLBUSY condition. This is not an error; you should wait briefly and try again.
- If you read multiple input files, you should delay your task briefly between closing one and opening the next, to avoid monopolizing the input thread and locking out other tasks that need it.

A remote application must route any files intended for a CICS transaction to a specific user name at the system where CICS resides. See Figure 94 on page 424 for an example of a CP command used by a VM system to do this. The figure also shows the EXEC CICS SPOOL commands you use to retrieve the data.

The CICS transaction issues the SPOOLOPEN command, specifying the writer name on the USERID parameter and optionally the class of output within the writer name. The normal response is:

1. No input for this external writer.
2. The single-thread is busy.
3. The file is allocated to you for retrieval, and is identified by the “token” returned by CICS. The token must be included on every SPOOL command for retrieving the data set.

In cases (1) and (2), the transaction should retry the SPOOLOPEN after a suitable interval, by restarting itself.

In case (3), the transaction should then retrieve the file with SPOOLREAD commands, and proceed to SPOOLCLOSE as rapidly as possible to release the path for other users. This is especially important for **input** from JES because the input path is **single-threaded**. When there is more than one transaction using the interface, their files can be differentiated by using different writer names or different classes within a single writer name. Furthermore, you should ensure that the transactions either terminate or wait for a short period between SPOOLCLOSE and a subsequent SPOOLOPEN. If you do not do this, one transaction can prevent others from using the interface.

JES exits

Both JES2 and JES3 provide a way of screening incoming files. For JES2, the TSO/E Interactive Data Transmission Facility Screening and Notification exit is used. The JES3 equivalent is the Validate Incoming Netdata File exit.

You should review any use your installation makes of these exits to ensure that files that are to be read using the CICS interface to JES are correctly processed.

Identifying spool files

Input spool files are identified by the USERID and CLASS options on the SPOOLOPEN INPUT command.

On input, the USERID is the name of a JES external writer. An external writer is a name defined to JES at JES startup representing a group of spool files that have the same destination or processing. For files that JES processes itself, an external writer is usually associated with a particular hardware device, for example, a printer. The names of these writers are reserved for JES use.

For the transfer of files between applications, as occurs when a CICS task reads a spool file, the only naming requirement is that the receiver (the CICS task) know what name the sender used, and that no other applications in the receiver's operating system use the same name for another purpose. To ensure that CICS tasks do not read spool files that were not intended for them, CICS requires that the external writer name that you specify match its own VTAM applid in the first four characters. Consequently, a job or system directing a file to CICS must send it to an external writer name that begins with the first four characters of the CICS applid.

JES categorizes the files for a particular external writer by a 1-character CLASS value. If you specify a class on your SPOOLOPEN INPUT command, you get the first (oldest) file in that class for the external writer you name. If you omit the class, you get the oldest file in any class for that writer. The sender assigns the class; 'A' is used when the sender does not specify a class.

On output, you identify the destination of a SPOOL file with both a NODE and a USERID value. The NODE is the name of the operating system (for example, MVS, VM) as that system is known to VTAM in the MVS system in which your CICS is executing).

The meaning of USERID varies with the operating system. In VM, it is a particular user; in MVS, it may be a JES external writer or another JES destination, a TSO user, or another job executing on that system. One such destination is the JES internal reader, which normally has the reserved name INTRDR. If you want to submit a job to an MVS system, you write a spool file to its internal reader. This file must contain all the JCL statements required to execute the job, in the same form and sequence as a job submitted through a card reader or TSO.

The following example shows a COBOL program using SPOOLOPEN for an internal reader. In this example, you must specify the NOCC option (to prevent use of the first character for carriage control) and use JCL record format.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
  01  OUTPUT-FIELDS.
      03  OUTPUT-TOKEN    PIC X(8)  VALUE LOW-VALUES.
      03  OUTPUT-NODE     PIC X(8)  VALUE 'MVSESA31'.
      03  OUTPUT-USERID   PIC X(8)  VALUE 'INTRDR  '.
      03  OUTPUT-CLASS    PIC X      VALUE 'A'.
PROCEDURE DIVISION.
  EXEC CICS SPOOLOPEN OUTPUT
      TOKEN(OUTPUT-TOKEN)
      USERID(OUTPUT-USERID)
      NODE(OUTPUT-NODE)
      CLASS(OUTPUT-CLASS)
      NOCC
      PRINT
      NOHANDLE
  END-EXEC.
```

Figure 96. An example of a COBOL program using SPOOL commands for an internal reader

OUTDESCR specifies a pointer variable to be set to the address of a field that contains the address of a string of parameters to the OUTPUT statement of MVS JCL.

The following example shows a COBOL program using the OUTDESCR operand:

```

WORKING-STORAGE SECTION.
01 F.
02 W-POINTER USAGE POINTER.
02 W-POINTER1 REDEFINES W-POINTER PIC 9(9) COMP.
01 RESP1 PIC 9(8) COMP.
01 TOKENWRITE PIC X(8).
01 ....
01 W-OUTDIS.
02 F PIC 9(9) COMP VALUE 43.
02 F PIC X(14) VALUE 'DEST(A20JES2 )'.
02 F PIC X VALUE ' '.
02 F PIC X(16) VALUE 'WRITER(A03CUBI)'.
02 F PIC X VALUE ' '.
02 F PIC X'11' VALUE 'FORMS(BILL)'.
LINKAGE SECTION.
01 DFHCOMMAREA PIC X.
01 L-FILLER.
02 L-ADDRESS PIC 9(9) COMP.
02 L-OUTDIS PIC X(1020).
PROCEDURE DIVISION.
    EXEC CICS GETMAIN SET(W-POINTER) LENGTH(1024)
        END-EXEC.
    SET ADDRESS OF L-FILLER TO W-POINTER.
    MOVE W-POINTER1 TO L-ADDRESS.
    ADD 4 TO L-ADDRESS.
    MOVE W-OUTDIS TO L-OUTDIS.
    EXEC CICS SPOOLOPEN
        OUTPUT
        PRINT
        RECORDLENGTH(1000)
        NODE('*')
        USERID('*')
        OUTDESCR(W-POINTER)
        TOKEN(TOKENWRITE)
        RESP(RESP1)
        NOHANDLE
    END-EXEC.
    EXEC CICS SPOOLWRITE
        .
        .
        .

```

Notes:

1. It is essential to code a GETMAIN command.
2. L-FILLER is not a parameter passed by the calling program. The BLL for L-FILLER is then substituted by the SET ADDRESS. The address of the getmained area is then moved to the first word pointed to by L-FILLER being L-ADDRESS (hence pointing to itself). L-ADDRESS is then changed by plus 4 to point to the area (L-OUTDIS) just behind the address. L-OUTDIS is then filled with the OUTDESCRIPTOR DATA. Hence W-POINTER points to an area that has a pointer pointing to the OUTDESCR data.

Examples of SPOOL commands

COBOL

```
#
#
# DATA DIVISION.
# WORKING-STORAGE SECTION.
# 01 RESP          PIC 9(8) BINARY.
# 01 RESP2         PIC 9(8) BINARY.
# 01 TOKEN         PIC X(8).
# 01 OUTLEN        PIC S9(8) BINARY VALUE +80.
# 77 OUTPRT        PIC X(80) VALUE
# 'SPOOLOPEN FUNCTIONING'.
# 01 PARMSPTR      POINTER.
# 01 PARMS-AREA.
#   03 PARMSLEN    PIC S9(8) BINARY VALUE 14.
#   03 PARMSINF    PIC X(14) VALUE
#   'WRITER(MYPROG)'.
#   03 PARMADDR    POINTER.
# PROCEDURE DIVISION.
#   SET PARMSPTR TO ADDRESS OF PARMS-AREA
#   SET PARMADDR TO PARMSPTR
#   SET PARMSPTR TO ADDRESS OF PARMADDR
#   EXEC CICS SPOOLOPEN OUTPUT
#       NODE ('*')
#       USERID ('*')
#       RESP(RESP) RESP2(RESP2)
#       OUTDESCR(PARMSPTR)
#       TOKEN(TOKEN)
#       END-EXEC
#   EXEC CICS SPOOLWRITE
#       FROM(OUTPRT)
#       RESP(RESP) RESP2(RESP2)
#       FLENGTH(OUTLEN)
#       TOKEN(TOKEN)
#       END-EXEC
#   EXEC CICS SPOOLCLOSE
#       TOKEN(TOKEN)
#       RESP(RESP) RESP2(RESP2)
#       END-EXEC.
#
```

PL/I

```
DCL
  RESP FIXED BIN(31),
  RESP2 FIXED BIN(31),
  TOKEN CHAR(8),
  OUTLEN FIXED BIN(31) INIT(80),
  OUTPRT CHAR(80) INIT('SPOOLOPEN FUNCTIONING'),
  PARMADDR POINTER,
  PARMSPTR POINTER;
DCL
  1 PARMS,
    2 PARMSLEN FIXED BIN(31) INIT(14),
    2 PARMSINF CHAR(14) INIT('WRITER(MYPROG)')
      ALIGNED;
PARMADDR=ADDR(PARMS);
PARMSPTR=ADDR(PARMADDR);
EXEC CICS SPOOLOPEN OUTPUT NODE('*') USERID('*')
      TOKEN(TOKEN) OUTDESCR(PARMSPTR) RESP(RESP)
      RESP2(RESP2);
EXEC CICS SPOOLWRITE FROM(OUTPRT) FLENGTH(OUTLEN)
      RESP(RESP) RESP2(RESP2) TOKEN(TOKEN);
EXEC CICS SPOOLCLOSE TOKEN(TOKEN) RESP(RESP)
      RESP2(RESP2);
```

C

```
#define PARMS struct _parms
PARMS
{
    int    parms_length;
    char   parms_info[200];
    PARMS * pArea;
};
PARMS ** parms_ptr;
PARMS  parms_area;
char userid[8]= "*";
char node[8]= "*";
char token[8];
long rcode1, rcode2;
/* These lines will initialize the outdescr area and
set up the addressing */
parms_area.parms_info[0]= '\0';
parms_area.pArea = &parms_area;
parms_ptr = &parms_area.pArea;
/* And here is the command with ansi carriage controls
specified and no class*/
EXEC CICS SPOOLOPEN OUTPUT
      NODE ( node )
      USERID ( userid )
      OUTDESCR ( parms_ptr )
      TOKEN ( token )
      ASA
      RESP ( rcode1 )
      RESP2 ( rcode2 );
```

ASSEMBLER

```
OUTPRT    DC      CL80'SPOOLOPEN FUNCTIONING'
PARMSPTR  EQU      6
RESP      DC      F'0'
RESP2     DC      F'0'
TOKEN     DS      2F
OUTPTR    DC      A(PARMSLEN)
PARMSLEN  DC      F'14'
PARMSINF  DC      C'WRITER(MYPROG)'
          LA      PARMSPTR,OUTPTR
          EXEC    CICS SPOOLOPEN OUTPUT OUTDESCR(PARMSPTR)
          NODE('*') USERID('*') RESP(RESP)
          RESP2(RESP2) TOKEN(TOKEN)
          EXEC    CICS SPOOLWRITE FROM(OUTPRT)
          TOKEN(TOKEN) RESP(RESP) RESP2(RESP2)
          EXEC    CICS SPOOLCLOSE TOKEN(TOKEN) RESP(RESP)
          RESP2(RESP2)
```

Chapter 30. CICS intercommunication

This chapter provides only a summary of what you need to consider when writing applications that communicate with other CICS systems. For further information, see the *CICS Intercommunication Guide*.

You can run application programs in a CICS intercommunication environment using one or more of the following:

Transaction routing

enables a terminal in one CICS system to run a transaction in another CICS system, see “Transaction routing” on page 434.

Function shipping

enables your application program to access resources in another CICS system, see “Function shipping” on page 434.

Distributed program link (DPL)

enables an application program running in one CICS region to link to another application program running in a remote CICS region, see “Distributed program link (DPL)” on page 435.

Asynchronous processing

enables a CICS transaction to start another transaction in a remote system and optionally pass data to it, see “Asynchronous processing” on page 446.

Distributed transaction processing (DTP)

enables a CICS transaction to communicate with a transaction running in another system. There are two interfaces available for DTP; command-level EXEC CICS and the SAA interface for DTP known as Common Programming Interface Communications (CPI Communications), see “Distributed transaction processing (DTP)” on page 446.

Common Programming Interface Communications (CPI-C)

provides DTP on APPC connections and defines an API that can be used on multiple system platforms, see “Common Programming Interface Communications (CPI Communications)” on page 446.

External CICS interface (EXCI)

enables a non-CICS program running in MVS to allocate and open sessions to a CICS system, and to issue DPL requests on these sessions. In CICS Transaction Server for z/OS, Version 3 Release 1, CICS supports MVS resource recovery services (RRS) in applications that use the external CICS interface. see “External CICS interface (EXCI)” on page 447.

The intercommunication aspects of the CICS Front End Programming Interface (FEPI) are not discussed in this book. See the *CICS Front End Programming Interface User's Guide* for details about FEPI.

Design considerations

If your application program uses more than one of these facilities, you obviously need to bear in mind the design considerations for each one. Also, if your program uses more than one intersystem session for distributed transaction processing, it must control each session according to the rules for that type of session.

Programming language

Generally speaking, you can use COBOL, C, C++, PL/I, or assembler language to write application programs that use CICS intercommunication facilities. There is, however, an exception. You can only use C, C++, or assembler language for DTP application programs that hold APPC unmapped conversations using the EXEC CICS API.

Transaction routing

Transactions that can be invoked from a terminal owned by another CICS system, or that can acquire a terminal owned by another CICS system during transaction initiation, must be able to run in a transaction routing environment.

Generally, you can design and code such a transaction just like one used in a local environment. However, there are a few restrictions related to basic mapping support (BMS), pseudoconversational transactions, and the terminal on which your transaction is to run. All programs, tables, and maps that are used by a transaction **must** reside on the system that owns the transaction. (You can duplicate them in as many systems as you need.)

Some CICS transactions are related to one another, for example, through common access to the CWA or through shared storage acquired using a GETMAIN command. When this is true, the system programmer must ensure that these transactions are routed to the same CICS system. You should avoid (where possible) any techniques that might create inter-transaction affinities that could adversely affect your ability to perform dynamic transaction routing.

To help you identify potential problems with programs that issue these commands, you can use the CICS Interdependency Analyzer. See the *CICS Interdependency Analyzer for z/OS User's Guide and Reference* for more information about this utility and Chapter 15, "Affinity," on page 221 for more information about transaction affinity.

When a request to process a transaction is transmitted from one CICS system to another, transaction identifiers can be translated from local names to remote names. However, a transaction identifier specified in a RETURN command is not translated when it is transmitted from the transaction-owning system to the terminal-owning system.

Function shipping

You code a program to access resources in a remote system in much the same way as if they were on the local system. You can use:

DL/I calls (EXEC DLI commands)

to access data associated with a remote CICS system.

File control commands

to access files on remote systems. Note that requests which contain the TOKEN keyword may not be function-shipped.

Temporary storage commands

to access data from temporary storage queues on remote systems.

Transient data commands

to access transient data queues on remote systems.

Three additional exception conditions can occur with remote resources. They occur if the remote system is not available (SYSIDERR), if a request is invalid (ISCINVREQ), or if the mirror transaction abends (ATNI for ISC connections and AZI6 for MRO).

Distributed program link (DPL)

The distributed program link function enables a CICS program (the client program) to call another CICS program (the server program) in a remote CICS region. There are several reasons why you might want to design your application to use distributed program link. Some of these are:

- To separate the end-user interface (for example, BMS screen handling) from the application business logic, such as accessing and processing data, to enable parts of the applications to be ported from host to workstation more readily
- To obtain performance benefits from running programs closer to the resources they access, and thus reduce the need for repeated function shipping requests
- To offer a simple alternative, in many cases, to writing distributed transaction processing (DTP) applications

There are several ways in which you can specify that the program to which an application is linking is remote:

1. By specifying the remote system name on a LINK command
2. By specifying the remote system name on the installed program resource definition⁴
3. By specifying the remote system name using the dynamic routing program (if the installed program definition specifies DYNAMIC(YES) or there is no installed program definition)⁴
4. By specifying the remote system name in a XPCREQ global user exit

The basic flow in distributed program link is described in the *CICS Intercommunication Guide*. The following terms, illustrated in Figure 97 on page 436, are used in the discussion of distributed program link:

Client region

The CICS region running an application program that issues a link to a program in another CICS region.

Server region

The CICS region to which a client region ships a link request.

Client program

The application program that issues a remote link request.

Server program

The application program specified on the link request, and which is executed in the server region.

4. By “installed program definition” we mean a program definition that has been installed statically, by means of autoinstall, or by an EXEC CICS CREATE command.

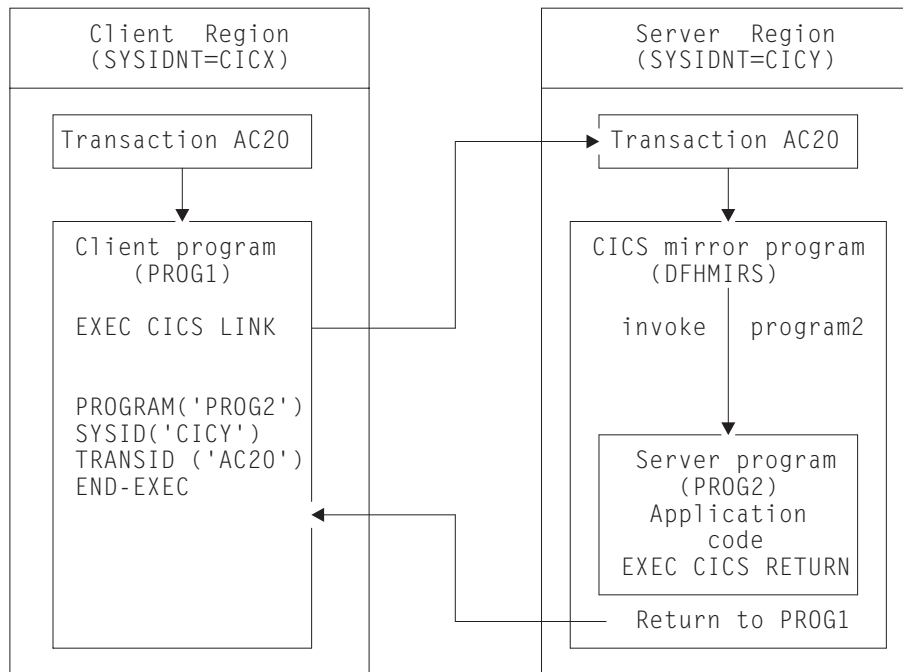


Figure 97. Illustration of distributed program link

Using the distributed program link function

The distributed program link function provides a number of options. You can specify:

- The name of the remote system (the server region).
- The name of the server program, if it is known by a different name in the server region.
- That you want to run the linked program locally, but restrict it to the distributed program link subset of the application programming interface (API) for testing purposes. (Server programs cannot use the entire CICS API when executed remotely; the restrictions are listed in Table 36 on page 445.)
- That the server program takes a syncpoint independently from the client.
- The name of the transaction you want the program to run under in the server region.
- The data length of the COMMAREA being passed.

A server program can itself issue a distributed program link and act as a client program with respect to the program it links to.

The options shown in Table 34 are used on the LINK command, and the options shown in Table 35 on page 437 are used in the PROGRAM resource definition in support of the distributed program link facility.

Table 34. Options on LINK command to support DPL

Keyword	Description
DATALENGTH	Specifies the length of the contiguous area of storage (from the start of the COMMAREA) that the application is sending to a server program.

Table 34. Options on LINK command to support DPL (continued)

Keyword	Description
SYSID	Specifies the name of the connection to the server region to which you want the client region to ship the program link request. Note: A remote SYSID specified on the LINK command overrides a REMOTESYSTEM name specified on the program resource definition or a sysid returned by the dynamic routing program.
SYNCONRETURN	Specifies that you want the server region to take a syncpoint on successful completion of the server program. Note: This option is unique to the LINK command and cannot be specified on the program resource definition.
TRANSID	Specifies the name of the transaction that the server region is to attach for execution of the server program. Note: TRANSID specified on the LINK command overrides any TRANSID specified on the program resource definition.

Note: Programming information, including the full syntax of the LINK command, is in the *CICS Application Programming Reference* manual, but note that for a distributed program link you cannot specify the INPUTMSG or INPUTMSGLEN options.

Table 35. Options in the PROGRAM resource definition to support DPL

Keyword	Description
REMOTESYSTEM	Specifies the name of the connection to the server region (SYSID) to which you want the client region to ship the program link request.
REMOTENAME	Specifies the name by which the program is known in the server region (if different from the local name).
DYNAMIC	Specifies whether the program link request can be dynamically routed. For detailed information about the dynamic routing of DPL requests, see the <i>CICS Recovery and Restart Guide</i> .
EXECUTIONSET	Specifies whether the program is restricted to the distributed program link subset of the CICS API. Note: This option is unique to the program definition and cannot be specified on the LINK command.
TRANSID	Specifies the name of the transaction that the server region is to attach for execution of the server program.

Examples of distributed program link

A COBOL example of a distributed program link command is shown in Figure 98 on page 438. The numbers down the right-hand side of the example refer to the numbered sections, following the figure, which give information about each option.

Important

If the SYSID option of the LINK command specifies the name of a remote region, any REMOTESYSTEM, REMOTENAME, or TRANSID attributes specified on the program definition or returned by the dynamic routing program have no effect.

```
EXEC CICS LINK PROGRAM('DPLPROG')      1
      COMMAREA(DPLPRO-DATA-AREA)      2
      LENGTH(24000)                   2
      DATALENGTH(100)                2
      SYSID('CICR')                  3
      TRANSID('AC20')                 4
      SYNCONRETURN                     5
```

Figure 98. COBOL example of a distributed program link

1. The program name of the server program

A program may have different names in the client and server regions. The name you specify on the LINK command depends on whether or not you specify the SYSID option.

If you specify the name of a remote region on the SYSID option of the LINK command, CICS ships the link request to the server region without reference to the REMOTENAME attribute of the program resource definition in the client region, nor to any program name returned by the dynamic routing program. *In this case, the PROGRAM name you specify on the LINK command must be the name by which the program is known in the server region.*

If you do not specify the SYSID option on the LINK command, or you specify the name of the local client region, *the PROGRAM name you specify on the LINK command must be the name by which the program is known in the client region.* CICS looks up the program resource definition in the client region.

Assuming that the REMOTESYSTEM option of the installed program definition specifies the name of a remote region, the name of the server program on the remote region is obtained from:

- The REMOTENAME attribute of the program definition
- If REMOTENAME is not specified, the PROGRAM option of the LINK command.

If the program definition specifies DYNAMIC(YES), or there is no installed program definition, the dynamic routing program is invoked and can accept or change the name of the server program.

2. The communication data area (COMMAREA)

To improve performance, you can specify the DATALENGTH option on the LINK command. This allows you to specify the amount of COMMAREA data you want the client region to pass to the server program. Typically, you use this option when a large COMMAREA is required to hold data that the server program is to return to the client program, but only a small amount of data needs to be sent to the server program by the client program, as in the example.

If more than one server program updates the same COMMAREA before it is passed back to the client program, use the DATALENGTH option to specify the length of the COMMAREA. Ensure that if any of the server programs use an XCTL command to pass the COMMAREA to the next server program, they specify the same length and address for it. This ensures that the original COMMAREA is returned to the client program. If a different length or address

are specified, the invoked program will receive a copy of the COMMAREA, rather than the original COMMAREA, and so the original COMMAREA will not be returned to the client program. “COMMAREA” on page 567 has more information about using COMMAREAs to pass data to other programs.

3. The remote system ID (SYSID)

You can specify the 4-character name of the server region to which you want the application region to ship a program link request using any of the following:

- The SYSID option of the LINK command
- The REMOTESYSTEM option of the program resource definition
- The dynamic routing program.

The rules of precedence are:

- a. If the SYSID option on the EXEC CICS LINK command specifies a remote CICS region, CICS ships the request to the remote region.

If the program definition specifies DYNAMIC(YES)—or there is no program definition—the dynamic routing program is invoked for notification only—it cannot re-route the request.

- b. If the SYSID option is not specified or specifies the same name as the local CICS region:

- 1) If the program definition specifies DYNAMIC(YES)—or there is no installed program definition—the dynamic routing program is invoked, and can route the request.

The REMOTESYSTEM option of the program definition, if specified, names the default server region passed to the dynamic routing program.

Note: If the REMOTESYSTEM option names a remote region, the dynamic routing program cannot route the request locally.

- 2) If the program definition specifies DYNAMIC(NO), CICS ships the request to the remote system named on the REMOTESYSTEM option. If REMOTESYSTEM is not specified, CICS runs the program locally.

The name you specify is the name of the connection definition installed in the client region defining the connection with the server region. (CICS uses the connection name in a table look-up to obtain the netname (VTAM APPLID) of the server region.) The name of the server region you specify can be the name of the client region, in which case the program is run locally.

If the server region is unable to load or run the requested program (DPLPROG in our example), CICS returns the PGMIDERR condition to the client program in response to the link request. Note that EIBRESP2 values are not returned over the link for a distributed program link request where the error is detected in the server region. For errors detected in the client region, EIBRESP2 values are returned.

You can also specify, or modify, the name of a server region in an XPCREQ global user exit program. See the *CICS Customization Guide* for programming information about the XPCREQ global user exit point.

4. The remote transaction (TRANSID) to be attached

The TRANSID option is available on both the LINK command and the program resource definition. This enables you to tell the server region the transaction identifier to use when it attaches the mirror task under which the server program runs. If you specify the TRANSID option, you must define the transaction in the server region, and associate it with the supplied mirror program, DFHMIRS. This option allows you to specify your own attributes on the transaction definition for the purpose of performance and fine tuning. For example, you could vary the task priority and transaction class attributes.

If the installed program definition specifies DYNAMIC(YES), or there is no installed program definition, the dynamic routing program is invoked and (provided that the SYSID option of the LINK command did not name a remote region) can change the value of the TRANSID attribute.

The order of precedence is:

- a. If the SYSID option of the LINK command specified a remote region, a TRANSID supplied on the LINK
- b. A TRANSID supplied by the dynamic routing program
- c. A TRANSID supplied on the LINK command
- d. The TRANSID attribute of the program definition.
- e. The mirror TRANSID, CSMI.

You are recommended to specify the transaction identifier of the client program as the transaction identifier for the server program. This enables any statistics and monitoring data you collect to be correlated correctly under the same transaction.

The transaction identifier used on a distributed link program request is passed to the server program as follows:

- If you specify your own transaction identifier for the distributed link program request, this is passed to the server program in the EIBTRNID field of the EIB.
- EIBTRNID is set to the TRANSID value as specified in the DPL API or server resource definition. Otherwise, it defaults to the client's transaction code, which is the same value that is in the client's EIBTRNID.

5. The SYNCONRETURN option for the server program

When you specify the SYNCONRETURN option, it means that the resources on the server are committed in a separate logical unit of work immediately before returning control to the client; that is, an implicit syncpoint is issued for the server just before the server returns control to the client. Figure 99 on page 441 provides an example of using distributed program link with the SYNCONRETURN option. The SYNCONRETURN option is intended for use when the client program is not updating any recoverable resources, for example, when performing screen handling. However, if the client does have recoverable resources, they are not committed at this point. They are committed when the client itself reaches a syncpoint or in the implicit syncpoint at client task end. You must ensure that the client and server programs are designed correctly for this purpose, and that you are not risking data integrity. For example, if your client program has shipped data to the server that results in the server updating a database owned by the server region, you only specify an independent syncpoint if it is safe to do so, and when there is no dependency on what happens in the client program. This option has no effect if the server program runs locally in the client region unless EXECUTIONSET(DPLSUBSET) is specified. In this case, the syncpoint rules governing a local link apply.

Without the SYNCONRETURN option, the client commits the logical unit of work for both the client and the server resources, with either explicit commands or the implicit syncpoint at task end. Thus, in this case, the server resources are committed at the same time as the client resources are committed. Figure 100 on page 441 shows an example of using distributed program link without the SYNCONRETURN option.

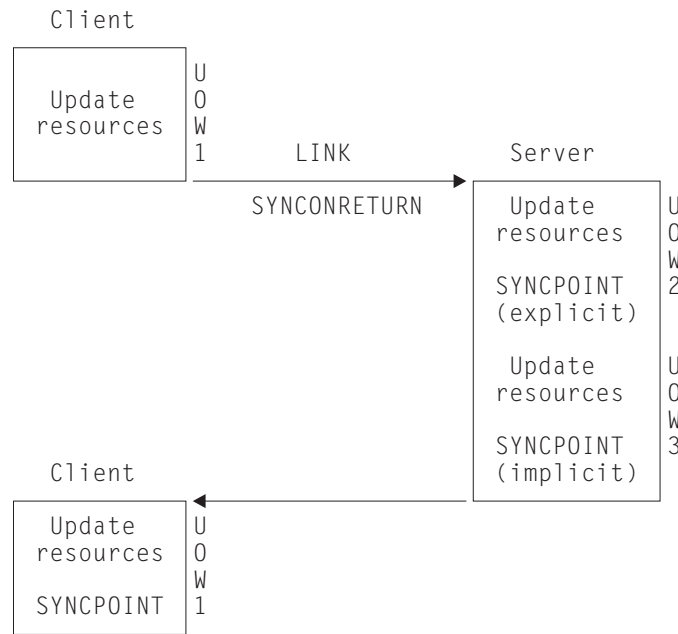


Figure 99. Using distributed program link with the SYNCONRETURN option

Note: This includes three logical units of work: one for the client and two for the server. The client resources are committed separately from the server.

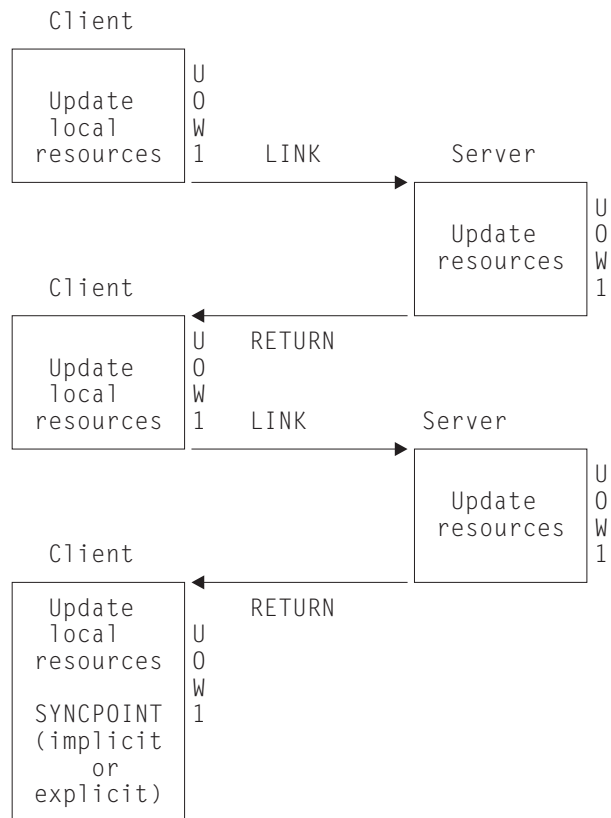


Figure 100. Using distributed program link without the SYNCONRETURN option

Note: The implicit or explicit syncpoint causes all client and server resources to be committed. There is only one logical unit of work because the client is responsible for determining when both the client and server resources are committed.

You need to consider the case when the client has a HANDLE ABEND command. When the client is handling abends in the server, the client gets control when the server abends. This is also true when the SYNCONRETURN option has been specified on the LINK command. In this case, it is recommended that the client issues an abend after doing the minimum of cleanup. This causes both the client logical unit of work and the server logical unit of work to be backed out.

Programming considerations for distributed program link

There are some factors you should consider when writing application programs that use distributed program link.

Issuing multiple distributed program links from the same client task

A client task cannot request distributed program links to a single CICS server region using more than one transaction code in a single client unit of work unless the SYNCONRETURN option is specified. It can issue multiple distributed program links to one CICS server system with the same or the default transaction code.

Sharing resources between client and server programs

The server program does not have access to the lifetime storage of tasks on the client, for example, the TWA. Nor does it necessarily have access to the resources that the client program is using, for example, files, unless the file requests are being function shipped.

Mixing DPL and function shipping to the same CICS system

Great care should be taken when mixing function shipping and DPL to the same CICS system, from the same client task. These are some considerations:

- A client task cannot function ship requests and then use distributed program link with the SYNCONRETURN option in the same session (same logical unit of work or system initialization parameter MROFSE=YES specified).. The distributed program link fails with an INVREQ response. In this case EIBRESP2 is set to 14.
- A client task cannot function ship requests and then use distributed program link with the TRANSID option in the same client logical unit of work. The distributed program link fails with an INVREQ response. In this case, EIBRESP2 is set to 15.
- Any function-shipped requests that follow a DPL request with the SYNCONRETURN option runs in a separate logical unit of work from the server logical unit of work.
- Any function-shipped requests running that follow a DPL request with the TRANSID option to the same server region runs under the transaction code specified on the TRANSID option, instead of under the default mirror transaction code. The function-shipped requests are committed as part of the overall client logical unit of work when the client commits.
- Any function-shipped requests running before or after a DPL request without the SYNCONRETURN or TRANSID options are committed as part of the overall client logical unit of work when the client commits.

See the *CICS Intercommunication Guide* for more information about function shipping.

Mixing DPL and DTP to the same CICS system

Care should be taken when using both DPL and DTP in the same application, particularly using DTP in the server program. For example, if you have not used the SYNCONRETURN option, you must avoid taking a syncpoint in the DTP partner which requires the DPL server program to syncpoint.

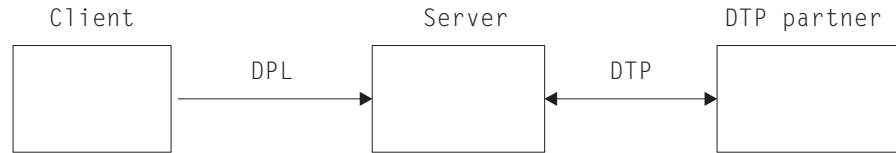


Figure 101. Example of mixing DPL and DTP

Restricting a program to the distributed program link subset

When a program executes as the result of a distributed program link, it is restricted to a subset of the full CICS API called the distributed program link subset. The commands that are prohibited in a server program are summarized in Table 36 on page 445.

You can specify, in the program resource definition only, that you want to restrict a program invoked by a local LINK command to this subset with the EXECUTIONSET(DPLSUBSET) option. The use of any prohibited commands can then be detected before an application program is used in a distributed environment. The EXECUTIONSET(DPLSUBSET) option should be used for very early testing purposes only, and should never be used in production.

When the server program is running locally the following considerations apply:

- If EXECUTIONSET(DPLSUBSET) is specified on the server program then the SYNCONRETURN option causes an implicit syncpoint to be taken in the local server program, prior to returning control to the client program. In this case, because the server program is running locally, both the client and server resources are committed. However, it should be noted that SYNCONRETURN is intended for use when the client has no recoverable resources.
- If EXECUTIONSET(FULLAPI) is specified on the server program, the SYNCONRETURN option is ignored.
- The TRANSID and DATALENGTH options are ignored when processing the local link, but the format of the arguments is checked, for example, the TRANSID argument cannot be all blank.

Determining how a program was invoked

The 2-byte values returned on the STARTCODE option of the ASSIGN command are extended in support of the distributed program link function enabling the server program to find out that it is restricted to the distributed program link subset. See the *CICS Application Programming Reference* manual for programming information about EXEC CICS commands.

Accessing user-related information with the ASSIGN command

The values returned with the USERID and OPID keywords of the ASSIGN command in the server program depend on the way the ATTACHSEC option is defined for the connection being used between the client CICS region and the server CICS region. For example, the system could be defined so that the server program could access the same USERID and OPID values as the client program or could access different values determined by the ATTACHSEC option.

If ATTACHSEC(LOCAL) is specified, the userid to which the OPID and USERID parameters correspond is one of the following, in the order shown:

1. The userid specified on the USERID parameter (for preset security) of the SESSIONS resource definition, if present
2. The userid specified on the SECURITYNAME parameter of the connection resource definition, if present and no preset security userid is defined on the sessions
3. The userid specified on the DFLTUSER system initialization parameter of the server region, if neither the sessions nor connection definitions specify a userid

If any value other than LOCAL is specified for ATTACHSEC, the signed-on userid is the one received in the function management header (FMH5) from the client region.

See the *CICS RACF Security Guide* for more information about link security and DPL security with MRO.

Another security-related consideration concerns the use of the CMDSEC and RESSEC options of the ASSIGN command. These are attributes of the transaction definition for the mirror transaction in the server region. They can be different from the definitions in the client region, even if the same TRANSID is used.

Exception conditions for LINK command

There are error conditions introduced in support of DPL.

Exception conditions returned to the client program: Condition codes returned to a client program describe such events as “remote system not known” or “failure to commit” in the server program. There are different reasons, identified by EIBRESP2 values, for raising the INVREQ and LENGERR conditions on a LINK command. The ROLLEDBACK, SYSIDERR, and TERMERR conditions may also be raised. See the *CICS Application Programming Reference* manual for programming information about these commands.

If the mirror transaction in the remote region fails, the application program that issued the DPL request can handle the abend of the mirror, and commit its own local resources, *only if both the following are true*:

1. The application program explicitly handles the abend caused by the mirror’s failure, and either:
 - Takes an implicit syncpoint by normal transaction termination
 - or Issues an explicit syncpoint request.
2. The remote mirror transaction performed no recoverable work within the scope of the application program’s unit of work. That is, the mirror was invoked only for a distributed program link (DPL) request with SYNCONRETURN.

In all other cases—that is, if the application program does not handle the abend, or the mirror does any recoverable work (for example, a file update, even to a non-recoverable file)—CICS forces the transaction to be backed out.

The PGMIDERR condition is raised on the HANDLE ABEND PROGRAM, LOAD, RELEASE, and XCTL commands if the local program definition specifies that the program is remote. This exception is qualified by an EIBRESP2 value of 9.

Exception conditions returned to the server program: The INVREQ condition covers the use of prohibited API commands. INVREQ is returned, qualified by an EIBRESP2 value of 200, to a server program if it issues one of the prohibited commands summarized in Table 36 on page 445. If the server program does not

handle the INVREQ condition, the default action is to abend the mirror transaction under which the server program is running with abend code ADPL.

For programming information about the DPL-related exception conditions, see the *CICS Application Programming Reference* manual.

Table 36. API commands prohibited in programs invoked by DPL

Command	Options
ASSIGN	ALTSCRNHT ALTSCRNWD APLKYBD APLTEXT BTRANS COLOR DEFSCRNHT DEFSCRNWD DELIMITER DESTCOUNT DESTID DESTIDLENG DS3270 DSSCS EWASUPP EXTDS FACILITY FCI GCHARS GCODES GMMI HILIGHT INPARTN KATAKANA LDCMNEM LDCNUM MAPCOLUMN MAPHEIGHT MAPLINE MAPWIDTH MSRCONTROL NATLANGINUSE NEXTTRANSID NUMTAB OPCLASS OPSECURITY OUTLINE PAGENUM PARTNPAGE PARTNS PARTNSET PS QNAME SCRNHT SCRNWD SIGDATA SOSI STATIONID TCTUALENG TELLERID TERMCODE TERMPRIORITY TEXTKYBD TEXTPRINT UNATTEND USERNAME USERPRIORITY VALIDATION
CONNECT PROCESS	all
CONVERSE	all
EXTRACT ATTRIBUTES	all
EXTRACT PROCESS	all
FREE	all
HANDLE AID	all
ISSUE	ABEND CONFIRMATION ERROR PREPARE SIGNAL PRINT ABORT ADD END ERASE NOTE QUERY RECEIVE REPLACE SEND WAIT
LINK	INPUTMSG INPUTMSGLEN
PURGE MESSAGE	all
RECEIVE	all
RETURN	INPUTMSG INPUTMSGLEN
ROUTE	all
SEND	CONTROL MAP PARTNSET TEXT TEXT(MAPPED) TEXT(NOEDIT) PAGE
SIGNOFF	all
SIGNON	all
START	TERMID
SYNCPPOINT	Can be issued in server region if SYNCONRETURN specified on LINK
WAIT TERMINAL	all
XCTL	INPUTMSG INPUTMSGLEN

The following commands are also restricted but can be used in the server region if SYNCONRETURN is specified on the LINK:

- CPIRR COMMIT
- CPIRR BACK
- EXEC DLI TERM
- CALL DLI TERM

Where only certain options are prohibited on the command, they are shown. All the APPC commands listed are prohibited only when they refer to the principal facility. One of these, the CONNECT PROCESS command, causes an error even if it refers to the principal facility in a non-DPL environment. It is included here because, if a CONNECT PROCESS command refers to its principal facility in a server program, the exception condition raised indicates a DPL error.

Asynchronous processing

The response from a remotely initiated transaction is not necessarily returned to the task that initiated the transaction, which is why the processing is referred to as **asynchronous**. Asynchronous processing is useful when you do not need or want to tie up local resources while having a remote request processed. For example, with online inquiry on remote databases, terminal operators can continue entering inquiries without having to wait for an answer to the first one.

You can start a transaction on a remote system using a START command just like a local transaction. You can use the RETRIEVE command to retrieve data that has been stored for a task as a result of a remotely issued START, CANCEL, SEND, or RECEIVE command, as if it were a local transaction.

Distributed transaction processing (DTP)

The main advantage of DTP is that it allows the two transactions to have exclusive control of a session and to “converse”. DTP is particularly useful when you need remote resources to be processed remotely or if you need to transfer data between systems. It also allows you to design very flexible and efficient applications. DTP can be used with either EXEC CICS or CPI Communications. You can use C, C++, and assembler language in DTP application programs that hold LU type 6.2 unmapped conversations using the EXEC CICS API as well as applications that use the CICS intercommunication facilities.

DTP can be used with a variety of partners, including both CICS and non-CICS platforms, as long as they support APPC. For further information about DTP, see the *CICS Distributed Transaction Programming Guide* and *CICS Family: Interproduct Communication* manuals.

Common Programming Interface Communications (CPI Communications)

CPI Communications provides an alternative API to existing CICS APPC support. CPI Communications provides DTP on APPC connections and can be used in COBOL, C, C++, PL/I, and assembler language.

CPI Communications defines an API that can be used in APPC networks that include multiple system platforms, where the consistency of a common API is seen to be of benefit.

The CPI Communications interface can converse with applications on any system that provides an APPC API. This includes applications on CICS platforms. You may use EXEC CICS APPC API commands on one end of a conversation and CPI Communications commands on the other.

CPI Communications requires specific information (side information) to begin a conversation with partner program. CICS implementation of side information is achieved using the partner resource which your system programmer is responsible for maintaining.

The application's calls to the CPI Communications interface is resolved by link-editing it with the CICS CPI Communications stub (DFHCPLC). You can find information about how to do this in "Including the CICS-supplied interface modules" on page 108.

The CPI Communications API is defined as a general call interface. The interface is described in the *Common Programming Interface Communications Reference* manual.

External CICS interface (EXCI)

The external CICS interface is an application programming interface that enables a non-CICS program (a client program) running in MVS to call a program (a server program) running in a CICS region and to pass and receive data by means of a communications area. The CICS program is invoked as if linked-to by another CICS program.

This programming interface allows a user to allocate and open sessions (pipes) to a CICS system and to pass distributed program link (DPL) requests over them. CICS interregion communication (IRC) supports these requests and each pipe maps onto one MRO session.

For programming information about EXCI, see the External Interfaces Guide.

A client program that uses the external CICS interface can operate multiple sessions for different users (either under the same or separate TCBs) all coexisting in the same MVS address space without knowledge of, or interference from, each other.

The external CICS interface provides two forms of programming interface:

- The EXCI CALL interface consists of six commands that allow you to:
 - Allocate and open sessions to a CICS system from non-CICS programs running under MVS
 - Issue DPL requests on these sessions from the non-CICS programs
 - Close and de-allocate the sessions on completion of the DPL requests
- The EXEC CICS interface provides:
 - A single composite command (LINK PROGRAM) that performs all six commands of the EXCI CALL interface in one invocation

The command takes the same form as the distributed program link command of the CICS command-level application programming interface.

CICS supports MVS resource recovery services (RRS) in applications that use the external CICS interface. This means that:

- The unit of work within which the CICS server program changes recoverable resources may now become part of the MVS unit of recovery associated with the EXCI client program.
- The CICS server unit of work may be committed when the server program returns control to the client or continues over multiple EXCI DPL calls, until the EXCI client decides to commit or backout the unit of recovery.

Part 6. Basic Mapping Support (BMS)

Chapter 31. Basic mapping support	453
BMS support levels	453
Minimum BMS	453
Standard BMS	453
Full BMS	454
A BMS output example	455
Chapter 32. Creating the map	459
Defining map fields: DFHMDF	459
Defining the map: DFHMDI	461
Defining the map set: DFHMDS	462
Writing BMS macros	463
Assembling the map	465
Physical and symbolic map sets	465
The SDF II alternative	466
Grouping maps into map sets	466
The Application Data Structure (ADS)	467
Using complex fields	467
Composite fields: the GRPNAME option	468
Repeated fields: the OCCURS option	469
Block data	469
Support for non-3270 terminals	470
Output considerations for non-3270 devices	470
Differences on input	471
Special options for non-3270 terminals	472
Device-dependent maps	472
Device dependent support: DDS	473
Finding out about your terminal	475
Chapter 33. Sending BMS mapped output	477
Acquiring and defining storage for the maps	477
BASE and STORAGE options	478
Initializing the output map	478
Moving the variable data to the map	479
Setting the display characteristics	479
Changing the attributes	480
Attribute value definitions: DFHBMSCA	481
Chapter 34. Using the SEND MAP command	483
SEND MAP control options	483
Other BMS SEND options: WAIT and LAST	484
Merging the symbolic and physical maps	484
MAPONLY option	484
DATAONLY option	485
The SEND CONTROL command	485
Building the output screen	485
What you start with	486
What is sent	486
Where the values come from	486
Outside the map	487
Using GDDM and BMS	488
Positioning the cursor	488
Sending invalid data and other errors	489

Output disposition options: TERMINAL, SET, and PAGING	489
Using SET	490
Chapter 35. Receiving mapped data	493
An input-output example	493
The symbolic input map	495
Programming mapped input.	496
Using the RECEIVE MAP command	496
Getting storage for mapped input.	497
Formatted screen input	497
Modified data	498
Upper case translation	499
Using the attention identifier	499
Using the HANDLE AID command	499
Finding the cursor	500
Processing the mapped input	500
Handling input errors	501
Flagging errors	501
Saving the good input	502
Rechecking.	502
Sending mapped output after mapped input.	503
MAPFAIL and other exceptional conditions	504
EOC condition.	504
Formatting other input.	505
Chapter 36. BMS logical messages	507
Building logical messages	507
The SEND PAGE command	508
RETAIN and RELEASE	508
The AUTOPAGE option	510
Terminal operator paging: the CSPG transaction	510
Logical message recovery	511
Chapter 37. Cumulative output — the ACCUM option	513
Floating maps: how BMS places maps using ACCUM	513
Page breaks: BMS overflow processing	514
Map placement rules	515
ASSIGN options for cumulative processing	517
Input from a composite screen.	517
Performance considerations.	517
Minimizing path length.	517
Reducing message lengths	518
Chapter 38. Text output.	521
The SEND TEXT command.	521
Text logical messages	521
Text pages	522
Text lines	523
Header and trailer format.	524
SEND TEXT MAPPED and SEND TEXT NOEDIT	525
Chapter 39. Message routing	527
Message destinations	527
Eligible terminals.	528
Destinations specified with OPCLASS only	528
OPCLASS and LIST omitted	529

Route list provided	529
Route list format	530
Message delivery	531
Undeliverable messages	532
Recoverable messages	532
Message identification	533
Programming considerations with routing	534
Routing and page overflow	534
Routing with SET	535
Interleaving a conversation with message routing	535
Chapter 40. The MAPPINGDEV facility	537
SEND MAP with the MAPPINGDEV option	537
RECEIVE MAP with the MAPPINGDEV option	538
Sample assembler MAPPINGDEV application	539
Chapter 41. Partition support	541
Uses for partitioned screens	542
Scrolling	542
Data entry	542
Lookaside	542
Data comparison	543
Error messages	543
Partition definition	543
3290 character size	544
Establishing partitioning	544
Partition options for BMS SEND commands	545
Determining the active partition	545
Partition options for BMS RECEIVE commands	546
ASSIGN options for partitions	546
Partitions and logical messages	546
Partitions and routing	547
Attention identifiers and exception conditions	547
Terminal sharing	548
Chapter 42. Support for special hardware	549
Logical device components	549
Defining logical device components	549
Sending data to a logical device component	550
LDCs and logical messages	550
LDCs and routing	550
10/63 magnetic slot reader	551
Field selection features	551
Trigger field support	551
Cursor and pen-detectable fields	552
Selection fields	553
Attention fields	553
BMS input from detectable fields	553
Outboard formatting	554

Chapter 31. Basic mapping support

Basic mapping support (BMS) is an application programming interface between CICS programs and terminal devices. BMS is one of two sets of commands for this purpose. The other one, terminal control, is described in Chapter 26, “Terminal control,” on page 365.

For many applications, BMS has several advantages. First, BMS removes device dependencies from the application program. It interprets **device-independent** output commands and generates **device-dependent** data streams for specific terminals. It also transforms incoming device-dependent data into device-independent format. These features eliminate the need to learn complex device data streams. They also allow you to use the same program for a variety of devices, because BMS determines the device information from the terminal definition, not from the application program.

Second, BMS separates the design and preparation of formats from application logic, reducing the impact of one on the other. Both of these features make it easier to write new programs and to maintain existing code.

This chapter describes:

- “BMS support levels”
- “A BMS output example” on page 455

BMS support levels

There are three levels of BMS support: minimum, standard, and full. Most installations use full BMS. If yours does, you can use all the features we describe in this chapter and not concern yourself with levels. If your installation uses minimum or standard BMS, you should note the features that require levels beyond yours. They are summarized here, and they are noted again whenever a facility that is not in minimum BMS is covered.

Minimum BMS

Minimum BMS supports all the basic functions for 3270 terminals, including everything described in our example and in the discussion of simple mapped output and mapped input.

Note: Minimum BMS has a substantially shorter path length than standard or full BMS. It is included in the larger versions and invoked as a kind of “fast path” on commands do not require function beyond what it provides. Specifically, it is used for SEND MAP and SEND CONTROL commands without the ACCUM, PAGING, SET, OUTPARTN, ACTPARTN, LDC, MSR, or REQID options, and for RECEIVE MAP commands, when your principal facility is a 3270 display or printer whose definition does not include outboard formatting. You can tell whether a particular BMS request used the fast path by looking at the CICS trace table. When fast path is used, the trace table contains duplicate entries for the BMS entry and exit code.

Standard BMS

Standard BMS adds:

- Support for terminals other than 3270s
- Text output commands

- Support for special hardware features: partitions, logical devices codes, magnetic slot readers, outboard formatting, and so on
- Additional options on the SEND command: NLEOM and FMHPARM

Standard BMS supports these terminals:

- Sequential terminals (composed of card readers, line printers, tape or disk)
- TCAM terminals (see “Terminal access method support” on page 366).

Note: In CICS TS 3.1, local TCAM terminals are not supported. The only TCAM terminals supported are remote terminals connected to a pre-CICS TS 3.1 terminal-owning region by the DCB (not ACB) interface of TCAM.

- TWX Model 33/35
- 1050
- 2740-1 (no buffer receive), 2740-2, 2741
- 2770
- 2780
- 2980, models 1, 2 and 4
- 3270
- 3600 (3601) LU
- 3650 (3653 and 3270 host conversational LUs)
- 3650 interpreter LU
- 3767/3770 interactive LU
- 3770 batch LU
- 3780
- LU type 4

Full BMS

Full BMS is required for:

- Sending BMS output other than directly to your own terminal (the SET and PAGING options, and BMS routing)
- Messages built cumulatively, with multiple BMS SEND commands (the ACCUM and PAGING options)

Some CICS platforms do not support all the features of BMS. Table 37 shows the approximate level of support in each, for general guidance. However, there are differences among platforms even at the same level, usually imposed by differences in execution environment. These are described in detail, by function, in *CICS Family: API Structure*. If your application may eventually move to another platform, or there is a chance that the end-user interface part of it may get distributed to one, you should consult that manual.

Table 37. BMS support across IBM platforms

Platform	BMS support
CICS OS/2	Minimum plus SEND TEXT of standard
CICS/400	Minimum plus SEND TEXT of standard
CICS/6000®	Minimum plus SEND TEXT of standard
CICS/VSE, CICS/DOS/VS	Full
CICS/ESA, CICS/MVS, CICS Transaction Server for OS/390, CICS Transaction Server for z/OS	Full

A BMS output example

To create a formatted screen, BMS takes a list of data items from a program and displays them on the screen (or printed page) according to a predefined format. It merges variable data supplied by the program with constant data in the format (titles, labels for variable fields, default values for these fields). It builds the data stream for the terminal to which you are writing, to show this merged data in the designated screen positions, with the proper attributes (color, highlighting, and so on). You do not have to know anything about the data stream, and you do not need to know much about the format to write the required CICS commands.

Note: For simplicity, this chapter is mainly concerned with display screens, but most of it applies equally to printers. Chapter 28, “CICS support for printing,” on page 407 discusses differences between displays and printers and covers additional considerations that apply to printing. Furthermore, the examples and discussion assume a standard 3270 terminal because BMS is designed to support the features of the 3270. Other terminals are discussed in “Support for non-3270 terminals” on page 470.

You define the formats, called **maps**, separately from the programs that use them. This allows you to reposition fields, change their attributes, and change the constant text without modifying your programs. If you add or remove variable data, of course, you need to change the programs which use the affected fields.

The basics of how this works are explained by an atypically simple example. In real life, requirements are always more complex, but this gives you the essentials without too much confusing detail. There are more realistic and complete BMS examples among the CICS sample applications. These programs are included in source form on the CICS distribution tape. More information can be found in the *Sample Applications Guide*.

This example assumes that you need to write the code for a transaction used in a department store that checks a customer’s balance before a charge sale is completed. The transaction is called a “quick check”, because all it does is check that the customer’s account is open and that the current purchase is permissible, given the state of the account. The program for the *output* part of this transaction gets an account number as input, and produces the screen shown in Figure 102 in response:

QCK	Quick Customer Account Check	
Account:	0000005	
Name:	Thompson	Chris
Max charge:	\$500.00	

Figure 102. Normal “quick check” output screen

The program uses the input account number to retrieve the customer’s record from the account file. From the information in this record, it fills in the account number and customer name in the map, and computes the maximum charge allowed from the credit limit, outstanding balance, and purchases posted after the last billing period. If the amount comes out negative, you are supposed to show a value of zero and add an explanatory message. You also need to alert the clerk if the charge card is listed as lost, stolen or canceled with a message as shown in Figure 103:

QCK	Quick Customer Account Check	
Account:	00000005	
Name:	Thompson	Chris
Max charge:	\$0.00	
STOLEN CARD - SECURITY NOTIFIED		

Figure 103. “Quick check” output screen with warning message

This message is to be highlighted, to draw the clerk’s attention to it.

The first thing you must do is define the screen. We explain how to do so for this particular map in Chapter 32, “Creating the map,” on page 459. For the moment, however, let us assume that one of the outputs of this process is a data structure like the one in Figure 104. (We show the COBOL-coded version of the structure, because we are using COBOL to code our examples. However, BMS produces the structure in any language that CICS supports.) The map creation process stores this source code in a library from which you copy it into your program.

The data names in this structure come from the map definition. You assign names

```

01 QCKMAP0.
  02 FILLER PIC X(12).
  02 FILLER PICTURE X(2).
  02 ACCTNOA PICTURE X.
  02 ACCTNOO PIC X(7).
  02 FILLER PICTURE X(2).
  02 SURNAMEA PICTURE X.
  02 SURNAMEO PIC X(15).
  02 FILLER PICTURE X(2).
  02 FNAMEA PICTURE X.
  02 FNAMEO PIC X(10).
  02 FILLER PICTURE X(2).
  02 CHGA PICTURE X.
  02 CHGO PIC $,$$0.00
  02 FILLER PICTURE X(2).
  02 MSGA PICTURE X.
  02 MSGO PIC X(30).

```

Figure 104. Symbolic map for “quick check”

to the fields that the program may have to change in any way. For our example, this category includes the fields where you display the account number, last name, first name, maximum charge, and explanatory message. It does not include any of the field labels or screen titles that never change, such as “Quick Customer Account Check” and “Account”.

Each field that you name on the screen generates several fields in the data structure, which are distinguished by a 1-character suffix added to the name you assigned in the map. Two appear here, the “A” suffix for the field attributes byte and the “O” suffix for the output data. If we were creating a map to use special device features like color and highlighting, or were using the map for input as well as output, there would be many more. We tell you about these other fields in “Setting the display characteristics” on page 479 and Chapter 35, “Receiving mapped data,” on page 493.

The key fields for this particular exercise are the ones suffixed with “O”. These are where you put the data that you want displayed on the screen. You use the “A” subfields if you want to change how the data is displayed. In our example, we use MSGA to highlight the message if our customer is using a dubious card.

Here is an outline of the code that is needed for the example. You have to copy in the data structure (Figure 104 on page 456) produced by creating the map, and the COPY QCKSET statement in the third line does this. (Ordinarily, you would use a copy statement for the account record format too. We show it partly expanded here so that you can see its contents.)

```

WORKING-STORAGE SECTION.
C COPY IN SYMBOLIC MAP STRUCTURE.
01 COPY QCKSET.
01 ACCTFILE-RECORD.
    02 ACCTFILE-ACCTNO                PIC S9(7).
    02 ACCTFILE-SURNAME              PIC X(15).
    02 ACCTFILE-FNAME                PIC X(10).
    02 ACCTFILE-CREDIT-LIM           PIC S9(7) COMP-3.
    02 ACCTFILE-UNPAID-BAL           PIC S9(7) COMP-3.
    02 ACCTFILE-CUR-CHGS             PIC S9(7) COMP-3.
    02 ACCTFILE-WARNCODE             PIC X.
    :
    :
PROCEDURE DIVISION.
    :
    EXEC CICS READ FILE (ACCT) INTO (ACCTFILE-RECORD) RIDFLD (CKNO)
        ... END-EXEC.
    MOVE ACCTFILE-ACCTNO TO ACCTNO0.
    MOVE ACCTFILE-SURNAME TO SURNAME0.
    MOVE ACCTFILE-FNAME TO FNAME0.
    COMPUTE CHGO = ACCTFILE-CREDIT-LIM - ACCTFILE-UNPAID-BAL
        - ACCTFILE-CUR-CHGS.
    IF CHGO < ZERO, MOVE ZERO TO CHGO
        MOVE 'OVER CHARGE LIMIT' TO MSGO.
    IF ACCTFILE-WARNCODE = 'S', MOVE DFHBMCRY TO MSGA
        MOVE 'STOLEN CARD - SECURITY NOTIFIED' TO MSGO
        EXEC CICS LINK PROGRAM('NTFYCOPS') END-EXEC.
    EXEC CICS SEND MAP ('QCKMAP') MAPSET ('QCKSET') END-EXEC.
    EXEC CICS RETURN END-EXEC.

```

Figure 105. BMS output example

Chapter 32. Creating the map

BMS provides three assembler language macro instructions (**macros**) for defining maps. This method of map definition is still widely used, and we are about to explain how to do it. However, there are also other products for creating maps which exploit the facilities of the display terminal to make the map creation process easier. They produce the same outputs as the BMS macros, generally with less programmer effort.

One of these is the Screen Definition Facility II (SDF II). SDF II allows you to build your screens directly from a display station, testing the appearance and usability as you go. You can find out more about SDF II in *Screen Definition Facility II General Introduction Part 1* and *Screen Definition Facility II General Introduction Part 2*.

The three assembler macros used to define BMS maps are:

DFHMDF

defines an individual field on a screen or page.

DFHMDI

defines a single map as a collection of fields.

DFHMSD

groups single maps into a map set.

The explanation of this process begins by telling you how to define individual fields. Then we explain how to go from the fields to a complete map, and from a map to a map set (the assembly unit). BMS is designed principally for 3270-type terminals, although it supports nearly all types. See Chapter 27, "The 3270 family of terminals," on page 387 for information on 3270 terms.

This chapter describes:

- "Defining map fields: DFHMDF"
- "Defining the map: DFHMDI" on page 461
- "Defining the map set: DFHMSD" on page 462
- "Writing BMS macros" on page 463
- "Assembling the map" on page 465
- "Using complex fields" on page 467
- "Block data" on page 469
- "Support for non-3270 terminals" on page 470
- "Device-dependent maps" on page 472

Defining map fields: DFHMDF

You should design the layout of your screen before you attempt to code any macros. After you have done that, you define each field on the screen (page) with a DFHMDF macro. In it, you indicate:

- The position of the field on the screen
- The length of the field
- The default contents (unless you always intend to provide them in the program)
- The **field** display attributes, governing whether and what the operator can key into the field, whether the cursor stops there, the intensity of the characters, and the initial state of the modified data tag
- For some terminals, **extended** display attributes, such as color, underlining, highlighting

- The name by which you refer to the field in your program, if you ever modify its contents or attributes

Fields that are referenced by the application must be allocated field names. The length of the field name and the characters that may be used to form field names must conform to the following rules. (Note that these rules apply to currently-supported compilers and assemblers.)

The characters used must be valid for names of assembler ordinary symbols. This character set consists of the alphabetic characters A - Z (upper or lower case), \$, #, @, numeric digits 0 - 9, and the underscore (_) character.

There is one exception to this rule. The hyphen (-) character may be used in field names provided that:

- The mapset is only used by application programs written in COBOL.
- The mapset is generated using the High Level Assembler.

The first character of the field name must be alphabetic, but the other characters can be any from the character set described above.

In addition, the characters used in field names must conform to the character set supported by the programming language of the application using the map. For example, if the application language is COBOL, you cannot use either the @ character or an underscore. You should refer to the appropriate Language Reference manual for information about these character sets.

The DFHMDF macro allows the length of field names to be from one through 30 characters. DFHMDF derives additional variable names by appending one of several additional characters to the defined name to generate a symbolic description map. These derived names may therefore be up to 31 characters in length. The assembler, PL/1, and C languages all support variable names of at least 31 characters. However the COBOL language only allows up to 30 characters, which means that field names used in maps must not exceed 29 characters for COBOL applications. For example, the following field definition is valid for all languages except COBOL:

```
ThisIsAnExtremelyLongFieldName DFHMDF LENGTH=10,POS=(2,1)
```

and the following field definition is only valid for COBOL:

```
Must-Not-Exceed-29-Characters DFHMDF LENGTH=10,POS=(2,1) "
```

Not all the options for field definition are described here; the rest are described in the *CICS Application Programming Reference* manual.

Figure 106 shows the field definitions for the map we considered in Figure 103 on page 456.

```

          DFHMD F POS=(1,1),LENGTH=3,ATTRB=(ASKIP,BRT),INITIAL='QCK'
          DFHMD F POS=(1,26),LENGTH=28,ATTRB=(ASKIP,NORM),                X
              INITIAL='Quick Customer Account Check'
          DFHMD F POS=(3,1),LENGTH=8,ATTRB=(ASKIP,NORM),INITIAL='Account: '
ACCTNO  DFHMD F POS=(3,13),LENGTH=7,ATTRB=(ASKIP,NORM)
          DFHMD F POS=(4,1),LENGTH=5,ATTRB=(ASKIP,NORM),INITIAL='Name: '
SURNAME DFHMD F POS=(4,13),LENGTH=15,ATTRB=(ASKIP,NORM)
FNAME   DFHMD F POS=(4,30),LENGTH=10,ATTRB=(ASKIP,NORM)
          DFHMD F POS=(5,1),LENGTH=11,ATTRB=(ASKIP,NORM),INITIAL='Max charge: '
CHG      DFHMD F POS=(5,13),ATTRB=(ASKIP,NORM),PICOUT='$,$$0.00'
MSG      DFHMD F LENGTH=20,POS=(7,1),ATTRB=(ASKIP,NORM)

```

Figure 106. BMS map definitions

1. The **POS** (position) parameter indicates the row and column position of the field, relative to the upper left corner of the map, position (1,1). It must be present. Remember that every field begins with a field attributes byte; POS defines the location of this byte; the contents of the field follow immediately to the right.
2. The **LENGTH** option tells how many characters long the field is. The length does *not* include the attributes byte, so each field occupies one more column than its LENGTH value. In the case of the first field in our map, for example, the attributes byte is in row 1, column 1, and the display data is in columns 2-4. Fields can be up to 256 characters long and can wrap from one line to another. (Take care with fields that wrap if your map is smaller than your screen. See “Outside the map” on page 487 for further information.)
3. The **ATTRB** (attributes) option sets the **field attributes** of the field, which we discussed in “3270 field attributes” on page 392. It is not required; BMS uses a default value of (ASKIP, NORM)—autoskip protection, normal intensity, modified data tag off—if you omit it. There are other options for each of the extended attributes, none of which was used in this map; these are described in “Setting the display characteristics” on page 479.
4. The **INITIAL** value for the field is not required either. You use it for label and title fields that have a constant value, such as ‘QCK’, and to assign a default value to a field, so that the program does not always have to supply a value.
5. The **PICOUT** option on the definition of the field CHG tells BMS what sort of PICTURE clause to generate for the field. It lets you use the edit facilities of COBOL or PL/I directly, as you move data into the map. If you omit PICOUT, and also the numeric (NUM) attribute, BMS assumes character data. Figure 104 on page 456 shows the effects of the PICOUT option for CHG and, in the other fields, its absence. You can omit the LENGTH option if you use PICOUT, because BMS infers the length from the picture.
6. The **GRPNAME** and **OCCURS** options do not appear in our simple example, because they are for more complex problems. GRPNAME allows you to subdivide a map field within the program for processing, and OCCURS lets you define adjacent, like map fields so that you can treat them as an array in the program. These options are explained in “Using complex fields” on page 467 after some further information about maps.

Defining the map: DFHMDI

After all the fields on your map are defined, you tell BMS that they form a single map by preceding them with a DFHMDI macro. This macro tells BMS:

- The name of the map
- The size, in rows and columns
- Where it appears on the screen (you can put several maps on one screen)
- Whether it uses 3270 extended display attributes and, if so, which ones

- The defaults for these extended attributes for fields where you have not assigned specific values on the DFHMDI macro
- Device controls associated with sending the map (such as whether to sound the alarm, unlock the keyboard)
- The type of device the map supports, if you intend to create multiple versions of the map for different types of devices (see “Device-dependent maps” on page 472)

The map name and size are the critical information on a DFHMDI macro but, for documentation purposes, you should specify your other options explicitly rather than letting them default. The DFHMDI macro for our example might be:

```
QCKMAP    DFHMDI SIZE=(24,80),LINE=1,COLUMN=1,CTRL=ALARM
```

We have named the map QCKMAP. This is the identifier we use in SEND MAP commands. It is 24 lines long, 80 columns wide, and starts in the first column of the first line of the display. We have also indicated that we want to sound the alarm when the map is displayed.

Defining the map set: DFHMSD

You need one more macro to create a map:DFHMSD, which defines a map set. Maps are assembled in groups called map sets. Typically you group all the maps used by a single transaction or several related transactions. (We discuss reasons for grouping maps further in “Grouping maps into map sets” on page 466.) A map set need not contain more than one map, incidentally, and in our simple example, the map set consists of just the “quick check” map.

One DFHMSD macro is placed in front of all the map definitions in the map set. It gives:

- The name of the map set
- Whether you are using the maps for output, input, or both
- Defaults for map characteristics that you did not specify on the DFHMDI macros for the individual maps
- Defaults for extended attributes that you did not specify in either the field or map definitions
- Whether you are creating physical or symbolic maps in the current assembly (see “Physical and symbolic map sets” on page 465)
- The programming language of programs that use the maps
- Information about the storage that is used to build the maps

Here’s the DFHMSD macro we need at the beginning of our example:

```
QCKSET    DFHMSD TYPE=MAP,STORAGE=AUTO,MODE=OUT,LANG=COBOL,TIOAPFX=YES
```

This map set definition tells BMS that the maps in it are used only for output, and that the programs using them are written in COBOL. It assigns the name QCKSET to the map set. TIOAPFX=YES causes inclusion of a 12-byte “prefix” field at the beginning of each symbolic map (you can see the effect in the second line in Figure 104 on page 456). You always need this filler in command language programs and you should specify it explicitly, as the default is sometimes omission. MAP and STORAGE are explained in Chapter 33, “Sending BMS mapped output,” on page 477.

You need another DFHMSD macro at the end of your map definitions, to tell the assembler that it has reached the end of last map in the map set:

Writing BMS macros

Because a BMS macro is an assembler language statement, you have to follow assembler syntax rules. We do not try to explain those in full here; you can find them in *Assembler H Version 2 Application Programming Language Reference* manual. Instead we give you a set of rules that work, although they are more restrictive than the actual rules.

1. Start names in column 1. Map and map set names may be up to seven characters long. The maximum length for field names (the DFHMDF macro) depends on the programming language. BMS creates labels by adding 1-character suffixes to your field names. These labels must not be longer than the target language allows, because they get copied into the program. Hence the limit for a map field name is 29 characters for COBOL, 30 for PL/I and Assembler H, and 7 for Assembler F. For C and C++, it is 30 if the map is copied into the program as an internal data object, and six if it is an external data object (see “Acquiring and defining storage for the maps” on page 477 for more information about copying the map).
2. Start the macro identifier in column 10, or leave one blank between it and the name if the name exceeds eight positions. For field definitions, the identifier is always DFHMDF; for map definitions, DFHMDF; and for the map set macros that begin and end the map set, DFHMSD.
3. The rest of the field description consists of keywords (like POS for the position parameter) followed by values. Sometimes a keyword does not have a value, but if it does, an equals sign (=) always separates the keyword from its value.
4. Leave one blank after your macro identifier and then start your keywords. They can appear in any order.
5. Separate keywords by one comma (no blanks), but do not put a comma after the last one.
6. Keywords can extend through column 71. If you need more space, stop after the comma that follows the last keyword that fits entirely on the line and resume in column 16 of the next line.
7. Initial values (the INITIAL, XINIT, and GINIT keywords) are exceptions to the rule, because they may not fit even if you start on a new line. Except when double-byte characters are involved, you can split them at any point after the first character of the initial value itself. When you split in this way, use all of the columns through 71 and continue in column 16 of the next line. Double-byte character set (DBCS) data is more complicated to express than ordinary single-byte (SBCS) data. See Step 12 if you have DBCS initial values.
8. Surround initial values by single quote marks. If you need a single quote *within* your text, use two successive single quotes (the assembler removes the extra one). Ampersands also have special significance to the assembler, and you use the same technique: use two ampersands where you want one, and the assembler removes the extra.
9. If you use more than one line for a macro, put a character (any one except a blank) in column 72 of all lines *except the last*.
10. If you want comments in your map, use comment lines between macros, not among the lines that make up a single macro. Comment lines have an asterisk in column 1 and a blank in column 72. Your comments can appear anywhere among columns 2-71.
11. Use upper case only, except for values for the INITIAL parameter and in comments.

12. **For initial values containing DBCS.** If you have initial data that is *entirely* DBCS, use the GINIT keyword for your data and specify the keyword PS=8 as well. If your data contains both DBCS and SBCS characters, that is, if it is **mixed**, use INITIAL and specify SOSI=YES. (We need to explain a third alternative, XINIT, because you may find it in code you are maintaining. You should use GINIT and INITIAL if possible, however, as XINIT is more difficult to use and your data is not validated as completely. XINIT can be used for either pure or mixed DBCS. XINIT with PS=8 follows the rules for GINIT, and XINIT with SOSI=YES follows those for INITIAL (mostly, at least). The main difference is that you express your data in hexadecimal with XINIT, but you use ordinary characters for GINIT and INITIAL.)

This is how you write DBCS initial values:

- You enclose your data with single quotes, as you do with the ordinary INITIAL parameter.
- You use two ordinary characters for each DBCS character in your constant (two pairs of hexadecimal digits with XINIT) and one for each SBCS character (one pair with XINIT).
- You bracket each DBCS character string with a shift-out (SO) character immediately preceding and a shift-in (SI) character immediately after. SO is hexadecimal X'0E', which appears as '<' on most keyboards, and SI is X'0F' ('>'). (XINIT with PS=8 is an exception; the SO/SI brackets are implied and you do not key them.) For example, all of these define the same initial value, which is entirely DBCS. (Ignore the LENGTH values for the moment; we explain those in a moment.)

```
GINIT='<D1D2D3D4D5>',PS=8,LENGTH=10
INITIAL='<D1D2D3D4D5>',SOSI=YES,LENGTH=12
XINIT='C4F1C4F2C4F3C4F4C4F5',PS=8,LENGTH=10
XINIT='0EC4F1C4F2C4F3C4F4C4F50F',SOSI=YES,LENGTH=12
```

- SBCS and DBCS sequences can follow each other in any combination with INITIAL (and XINIT with SOSI=YES). If we add 'ABC' in front of the DBCS string in the previous example, and 'def' following the string, we have:

```
INITIAL='ABC<D1D2D3D4D5>def',SOSI=YES,LENGTH=18
XINIT='C1C2C30EC4F1C4F2C4F3C4F4C4F50F848586',SOSI=YES,LENGTH=18
```

- To calculate the length of your initial value, count two for each DBCS character and one for each SBCS character, whether you express them in ordinary characters or hexadecimal pairs. With GINIT (and XINIT with PS=8), you do not count the SO and SI characters, but with INITIAL (and XINIT with SOSI=YES), you add one for each SO and for each SI. (Note the different LENGTH values for the same constants in the examples above.) In all cases, your LENGTH value must not exceed 256.
- For GINIT and INITIAL, if your constant does not fit on one line, you use "extended" continuation rules, which are a little different from the ones described earlier. With extended continuation, you can stop after any full character (SBCS character, DBCS pair, or the SI ending a DBCS string) within your initial value. If you are in the middle of a DBCS string, add an SI (the SOs and SIs on one line must balance). Then fill out the line through column 72 with a continuation character. Any character will do, so long as it is different from the last meaningful character on the line.

If you have stopped within a DBCS string, put an SO character in column 16 of the next line and resume in 17; otherwise just resume in 16, thus:

```
GXMPL1  DFHMD  POS=(02,21),LENGTH=20,PS=8,GINIT='<D1D2D3D4D5D6>*****  
          <D7D8D9D0>'  
IXMPL1  DFHMD  POS=(02,21),LENGTH=23,PS=8,INITIAL='abc<D1D2D3D4>ABC**  
          DEFGHIJ'
```

You cannot use extended continuation with XINIT; use the rules described in Step 7.

- If your LENGTH specification exceeds the length of the initial value you provide, the value is filled out on the right with DBCS blanks to your LENGTH value if you have used GINIT (or XINIT with PS=8). If you have used INITIAL, the fill character is an SBCS blank if the last part of the constant was SBCS, a DBCS blank if the last part was DBCS. If you use XINIT with SOSI=YES, the fill character is always an SBCS blank.

Assembling the map

Before you start coding, you must assemble and link edit your map set. You usually have to assemble twice, to create the map set in two different forms. The TYPE option in the DFHMSD macro tells the assembler the form to produce in any particular assembly.

Physical and symbolic map sets

A TYPE=MAP assembly, followed by a link-edit, produces a load module called the **physical map set**. The physical map set contains format information in encoded form. CICS uses it at execution time for constant fields and to determine how to merge in the variable data from the program. The physical map set normally is stored in the same library as your application programs, and it requires a MAPSET resource definition within CICS, just as a program requires a PROGRAM resource definition.

The output of a TYPE=DSECT assembly is a series of data structures, collectively called the **symbolic map set**, coded in the source language specified in the LANG option. There is a structure for each map used for input, called the **symbolic input map**, and one for each map used for output, called the **symbolic output map**.

Symbolic map sets are used at compile (assembly) time. You copy them into your program, and they allow you to refer to the fields in the maps by name and to pass the variable data in the form dictated by the physical map set. We have already shown you an example of a symbolic output map in COBOL (see Figure 104 on page 456) and used it in the example code. Symbolic map sets are usually stored in the library your installation defines for source code that gets copied into programs. Member names are usually the same as the map set names, but they need not be.

You need the TYPE=DSECT assembly before you compile or assemble your program. You can defer the TYPE=MAP assembly and link-edit until you are ready to test, because the physical map set is not used until execution time. However, because you must do both eventually, many installations provide a catalogued procedure to do this automatically; the procedure copies the source file for the map set and processes it once using TYPE=MAP and again using TYPE=DSECT. You also can use the SYSPARM option in your assembler procedure to override the TYPE value in a particular assembly. See the *Assembler H Version 2 Application Programming Language Reference* manual for a description of SYSPARM in

connection with map assemblies, and Chapter 10, “Installing map sets and partition sets,” on page 125 for more information about assembling maps.

Notes:

1. The fact that symbolic map sets are coded in a specific language does not prevent you from using the same map in programs coded in different languages. You simply assemble with TYPE=DSECT for each LANG value you need, taking care to store the outputs in different libraries or under different names. The LANG value does not affect the TYPE=MAP assembly, which need be done only once.
2. If you modify an existing map in a way that affects the symbolic map, you *must recompile (reassemble)* any programs using it, so that the compilation uses the symbolic structure that corresponds to the new physical structure. Changes to unnamed map fields do not affect the symbolic map, but addition, deletion, rearrangement, and length changes of named fields do. (Rearrangement refers to the DFHMDF macros; the order of the fields on the screen does not affect the symbolic map, although it is more efficient to have the DFHMDF macros in same order as the fields on the screen.) So make changes to the DSATTS option in the map definition—this option states the extended attributes you may want to change by program. It is always safest to recompile, of course.

The SDF II alternative

None of these assembly or link-edit steps is required if you use the IBM licensed program Screen Definition Facility II. SDF II produces creates both the symbolic map set and the physical map set in the final step of the interactive map creation process. SDF II can run under either MVS (Program 5665-366) or VM (5664-307). Refer to the *Screen Definition Facility II Primer for CICS/BMS Programs*, the *Screen Definition Facility II General Introduction Part 1*, and the *Screen Definition Facility II General Introduction Part 2*. More information can be found in the *Screen Definition Facility II General Information* and *Screen Definition Facility II Primer for CICS/BMS Programs*.

Grouping maps into map sets

Because they are assembled together, all of the physical maps in a map set constitute a single load module. BMS gains access to all of them with a single load request, issued on the first use of the map set in the task. No further loads are required unless you request a map in a different set, in which case BMS releases the old map set and loads the new one. If you go back to the first map set subsequently, it gets loaded again. Loading and deleting does not necessarily involve I/O, but you should consider the path length when grouping your maps into map sets. Generally, if maps are used together, they should be in the same map set; those not used together should be in different map sets.

The limit to the number of maps in a set is 9 998, but you should also keep the size of any given load module reasonable. So you might keep infrequently used maps separate from those normally used in a given process.

Similarly, all of the symbolic maps for a map set are in a single symbolic structure. This affects the amount of storage you need while using the maps, as explained in “BASE and STORAGE options” on page 478. Depending on the programming language, it also may affect high-level names, and this may be a reason for separating or combining maps as well.

The Application Data Structure (ADS)

The symbolic map generated by the BMS macros is also known as the application data structure (ADS).

Physical maps produced by CICS Transaction Server for z/OS, Version 3 Release 1 also include an ADS descriptor in the output load module. This is provided to allow interpretation of the BMS Application Data Structure (the structure used by the application program for the data in SEND and RECEIVE MAP requests), without requiring your program to include the relevant DSECT or copybook at compile time.

The ADS descriptor contains a header with general information about the map, and a field descriptor for every field that appears in the ADS (corresponding to every named field in the map definition macro). It can be located in the mapset from an offset field in DFHMAPDS.

The ADS descriptor is generated for all maps. You can choose to map the long or short form of the ADS by specifying the DSECT=ADSIADSL option. The default is ADS, the short (normal) form. The long form of the ADS aligns all fields on 4-byte boundaries and is required for some interfaces with other products, such as MQSeries®.

Map sets generated with CICS releases before CICS Transaction Server for OS/390 Release 2 do not contain the ADS descriptor.

The format of the ADS descriptor is contained in the following copybooks:

Table 38. ADS descriptor copybooks

Language	Copybook
Assembler	DFHBRARD
C	DFHBRARH
PL/I	DFHBRARL
COBOL	DFHBRARO

For further information about the ADS descriptor, see the *CICS External Interfaces Guide*.

If you need to reassemble maps but have no access to the source, a utility program, DFHBMSUP, is provided in CICS Transaction Server for z/OS, Version 3 Release 1 to recreate BMS macro source from a mapset load module.

See the *CICS Operations and Utilities Guide* for more information about DFHBMSUP.

Using complex fields

The symbolic maps we have shown so far consisted of a fixed set of fields for each named map field (the A and O subfields, and so on, in Figure 104 on page 456). Such fields are the most common, but BMS provides two options for field definition which produce slightly different structures, to account for two common programming situations.

Composite fields: the GRPNAME option

Sometimes, you have to refer to subfields within a single field on the display. For example, you may have a date field that appears on the screen like this:

03-17-92

It is one field on the screen (with one attributes byte, just before the digit “0”), but you must be able to manipulate the month, day, and year components separately in your program.

You can do this with a “group field”, using the GRPNAME option of the DFHMDF macro. To create one, you code a DFHMDF macro for each of the component subfields; each definition names the same group field in the GRPNAME option. To define the date above as a group field starting at the beginning of line 10, for example, we would write:

```
MO      DFHMDF POS=(10,1),LENGTH=2,ATTRB=BRT,GRPNAME=DATE
SEP1    DFHMDF POS=(10,3),LENGTH=1,GRPNAME=DATE,INITIAL='-'
DAY     DFHMDF POS=(10,4),LENGTH=2,GRPNAME=DATE
SEP2    DFHMDF POS=(10,6),LENGTH=1,GRPNAME=DATE,INITIAL='-'
YR      DFHMDF POS=(10,7),LENGTH=2,GRPNAME=DATE
```

These definitions produce the following in the symbolic output map:

```
02 DATE.
03 FILLER PICTURE X(2).
03 MOA    PICTURE X.
03 MOO    PIC X(2).
03 SEP1   PIC X(1).
03 DAO    PIC X(2).
03 SEP2   PIC X(1).
03 YRO    PIC X(2).
```

Several rules must be observed when using a group field:

- There is only one attributes byte; it precedes the whole group field and applies to the whole field. You specify it just once, on the DFHMDF macro for the first subfield, MO here.
- Because there is only one attributes byte, the cursor behaves as if the group field were a single field. In our example, the cursor does not move from the last position of month to the first of day, or day to year, skipping over the hyphens. This is because the group really is a single field as far as the hardware goes; it is subdivided only for program access to the component subfields.
- Although subfields after the first do not have an attributes byte, you define the POS option as if they did, as shown in the example. That is, POS points to one character before the subfield begins, and can overlap the last character of the previous subfield, as occurs in our example.
- Although all the component subfields are adjacent in this example, they do not have to be. There can be gaps between the subfields, provided you do not define any other field in the gap. The group field spans all the columns from its first subfield to its last, and you must put the component DFHMDF macros in the order the subfields appear on the screen. The group ends with the first DFHMDF macro that does not specify its name.
- You must assign a field name to every subfield, even if you do not intend to refer to it (as we did in the SEP1 and SEP2 subfields in the example).
- You cannot use the OCCURS option (explained in the next section) for a group field or any of its components.

Repeated fields: the OCCURS option

Sometimes a screen contains a series of identical fields that you want to treat as an array in your program. Suppose, for example, that you need to create a display of 40 numbers, to be used when a clerk assigns an unused telephone number to a new customer. (The idea is to give the customer some choice.) You also want to highlight numbers which have been in service recently, to warn the customer of the possibility of calls to the previous owner.

You can define the part of your screen which shows the telephone numbers with a single field definition:

```
TELNO    DFHMDF POS=(7,1),LENGTH=9,ATTRB=NORM,OCCURS=40
```

This statement generates 40 contiguous but separate display fields, starting at position (7,1) and proceeding across the rows for as many rows as required (five, in our case). We have chosen a length that (with the addition of the attributes byte) divides the screen width evenly, so that our numbers appear in vertical columns and are not split across row boundaries. The attributes you specify, and the initial value as well, apply to each field.

The description of these fields in the symbolic map looks like this in COBOL:

```
02 TELNOG    OCCURS 40.  
03 FILLER PICTURE X(2).  
03 TELNOA    PICTURE X.  
03 TELNOO    PIC X(9).
```

This structure lets you fill the map from an array in your program (or any other source) as follows:

```
        PERFORM MOVENO FOR I FROM 1 THROUGH 40.  
        ...  
MOVENO.  
        MOVE AVAIL-NO (I) TO TELNOO (I).  
        IF DAYS-SINCE-USE (I) < 90, MOVE DFHMBRY to TELNOA (I).
```

(DFHMBRY is a CICS-supplied constant for setting the field intensity to bright; we explain more in “Attribute value definitions: DFHBMSCA” on page 481.)

Labels for OCCURS fields vary slightly for the different languages that CICS supports, but the function is the same.

Each element of an array created by the OCCURS option is a single map field. If you need to repeat a series of fields (an array of structures, in other words), you cannot use OCCURS. To use such an array in a program, you must define all of the fields individually, without OCCURS, to produce the necessary physical map. Then you can modify the resulting symbolic map, replacing the individual field definitions with an array whose elements are the structure you need to repeat. You must ensure that the revised symbolic map has exactly the same field structure as the original, of course. An alternative is to use SDF II, which allows you to define such an array directly.

Block data

BMS provides an alternate format for the symbolic map, called block data format, that may be useful in specific circumstances. In block data format, the symbolic output map is an image of the screen or page going to the terminal. It has the customary field attributes (A) and output value (O) subfields for each named map field, but the subfields for each map field are separated by filler fields such that their

displacement in the symbolic map structure corresponds to their position on the screen. There are no length subfields, and symbolic cursor positioning is unavailable as a consequence.

For example, the symbolic map for the “quick check” screen in Figure 103 on page 456 would look like this in block data format (assuming a map 80 columns wide). Compare this with the normal “field data”. format (in Figure 104 on page 456) from the same map definition.

You can set only the field attributes in the program; BMS ignores the DSATTS

```
01 QCKMAP0.
  02 FILLER PIC X(12).           <---TIOAPFX still present
  02 FILLER PICTURE X(192).      <---Spacer
  02 ACCTNOA PICTURE X.         <---Position (3,13)
  02 ACCTNOO PIC X(7).
  02 FILLER PICTURE X(72).      <---Spacer
  02 SURNAMEA PICTURE X.        <---Position (4,13)
  02 SURNAMEO PIC X(15).
  02 FNAMEA PICTURE X.          <---Position (4,30),
  02 FNAMEO PIC X(10).
  02 FILLER PICTURE X(52).      <---Spacer
  02 CHGA PICTURE X.            <---Position (5,13)
  02 CHGO PIC $,$$0.00
  02 FILLER PICTURE X(139).     <---Spacer
  02 MSGA PICTURE X.            <---Position (7,1).
  02 MSGO PIC X(30).
```

Figure 107. Symbolic map for “quick check” in block data format

option in the map and does not generate subfields for the extended attributes in block data format. You can use block data for input as well. The input map is identical in structure to the output map, except that the flag (F) replaces the field attributes (A) subfield, and the input (I) replaces the output (O) subfield, as in field format.

Block data format may be useful if the application program has built or has access to a printer page image which it needs to display on a screen. For most situations, however, the normal field data format provides greater function and flexibility.

Support for non-3270 terminals

Minimum BMS supports only 3270 displays and printers. This category includes the 3178, 3290, 8775 and 5520, LU type 2 and LU type 3 devices, and any other terminal that accepts the 3270 data stream. The *IBM 3270 Information Display System Data Stream Programmer's Reference* manual contains a full list. **Standard** BMS expands 3270 support to SCS printers (3270 family printers *not* using the 3270 data stream) and all of the terminal types listed in Table 39 on page 474. See “Non-3270 CICS printers” on page 412 for more information about BMS and SCS datastreams.

Because of functional differences among these terminal types, it is not possible to make BMS work in exactly the same way for each of them. The sections which follow outline the limitations in using BMS on devices which lack the hardware basis for certain features.

Output considerations for non-3270 devices

Because BMS separates the device-dependent content of the output data stream from the logical content, there are only a few differences between 3270 and non-3270 devices that you need to consider in creating BMS output.

The primary difference between 3270 and non-3270 devices is that the 3270 is **field-oriented**, and most others are not. Consequently, there are neither field attributes nor extended attributes associated with output fields sent to non-3270 terminals. BMS can position the output in the correct places, field by field, but the field structure is not reflected in the data stream. BMS can even emulate some field attributes on some terminals (it may underline a highlighted field, for example), but there is no modified data tag, no protection against keying into the field, and so on.

If you specify attributes on output that the terminal does not support, BMS simply ignores them. You do not need to worry about them, provided the output is understandable without the missing features.

Differences on input

The absence of field structure has more impact on input operations, because many of the features of BMS depend on the ability to read—by field—only those fields that were modified. If the hardware does not provide the input field-by-field with its position on the screen, you must provide equivalent information.

You can do this in either of two ways. The first is to define a field-separator sequence, one to four characters long, in the FLDSEP option of the map definition. You place this sequence between each field of your input and supply the input fields in the same order as they appear on the screen or page. You must supply every field on the screen, up to the last one that contains any data. If there is no input in a field, it can consist of only the terminating field-separator sequence. On hardcopy devices, input cannot overlay the output because of paper movement. On displays that emulate such terminals, the same technique is generally used. Input fields are entered in an area reserved for the purpose, in order, separated by the field-separator sequence.

The second method is to include control characters in your input. If you omit the FLDSEP option from your map, BMS uses control characters to calculate the position of the data on the “page” and maps it accordingly. The control characters that BMS recognizes are:

NL	new line	X'15'
IRS	interchange record separator	X'1E'
LF	line feed	X'25'
FF	form feed	X'0C'
HT	horizontal tab	X'05'
VT	vertical tab	X'0B'
CR	carriage return	X'0D'
RET	return on the TWX	X'26'
ETB	end text block	X'26'
ESC	escape, for 2780	X'27'

When you read data of this kind with a RECEIVE MAP command, there are some differences from true 3270 input:

- The flag byte (F subfield) is not set; it contains a null. You cannot determine whether the operator erased the field or whether the cursor was left in the field.
- You cannot preset a modified data tag on output to ensure that a field is returned on input.

Special options for non-3270 terminals

BMS provides some additional formatting options for non-3270 devices, to take advantage of device features that shorten the data stream. These include:

- Vertical and horizontal tabs. You can position your output with horizontal and vertical tab orders if the device supports them. The tab characters are defined by the HTAB and VTAB options in the map set definition. When you want to position to the next horizontal tab, you include the HTAB character in your data; you position to the next vertical tab by supplying the VTAB character in your data. BMS translates these characters to the tab sequence required by your particular principal facility.

Before you use tabs in BMS output, your task or some earlier task at the same terminal must have set the tabs in the required positions. This is usually done with a terminal control SEND command, described in “Using data transmission commands” on page 370.

- Outboard formatting. Some logical units can store format information and participate in the formatting process. This allows BMS to send much less data (essentially the symbolic map contents) and delegate the work of merging the physical and symbolic maps to the logical unit. See “Outboard formatting” on page 554 for details.
- NLEOM (new line, end of message). Standard BMS also gives you the option of requesting that BMS format your output with blanks and new-line (NL) characters rather than 3270 buffer control orders. This technique gives you more flexibility in page width settings on printers, as explained in “NLEOM option” on page 410.

Device-dependent maps

Because the position, default attributes, and default contents of map fields appear only in the physical map and not in the symbolic map, you can use a single program to build maps that contain the same variable information but different constant information in different arrangements on the screen. This is very convenient if the program you are writing must support multiple devices with different characteristics.

You do this by defining multiple maps with the same names but different attributes and layout, each with a different suffix.

Suppose, for example, that some of the clerks using the “quick update” transaction use 3270 Model 2s (as we have assumed until now), and the rest use a special-purpose terminal that has only 3 rows and 40 columns. The format we designed for the big screen will not do for the small one, but the information will fit if we rearrange it:

We need the following map definition:

```
QUP Quick Account Update:
Current charge okay; enter next
Acct: _____ Charge: $ _____
```

Figure 108. “Quick update” for the small screen


```

QUPSET  DFHMSD TYPE=MAP,STORAGE=AUTO,MODE=INOUT,LANG=COBOL,SUFFIX=9
QUPMAP  DFHMDI SIZE=(3,40),LINE=1,COLUMN=1,CTRL=FREEKB
        DFHMDF POS=(1,1),LENGTH=24,ATTRB=(ASKIP,BRT),                      X
        INITIAL='QUP Quick Account Update'
MSG      DFHMDF LENGTH=39,POS=(2,1),ATTRB=(ASKIP,NORM)
        DFHMDF POS=(3,1),LENGTH=5,ATTRB=(ASKIP,NORM),                      X
        INITIAL='Acct:'
ACCTNO   DFHMDF POS=(3,11),LENGTH=6,ATTRB=(UNPROT,NUM,IC)
        DFHMDF POS=(3,18),LENGTH=1,ATTRB=(ASKIP),INITIAL=' '
        DFHMDF POS=(3,20),LENGTH=7,ATTRB=(ASKIP,NORM),INITIAL='Charge:'
CHG       DFHMDF POS=(3,29),LENGTH=7,ATTRB=(UNPROT,NORM),PICIN='$$$$0.00'
        DFHMDF POS=(3,37),LENGTH=1,ATTRB=(ASKIP),INITIAL=' '
        DFHMSD TYPE=FINAL

```

Figure 109. Map definition

The symbolic map set produced by assembling this version of the map is identical to the one shown in “An input-output example” on page 493, because the fields with names have the same names and same lengths, and they appear in the same order in the map definition. (They do not need to appear in the same order on the screen, incidentally; you can rearrange them, provided you keep the definitions of named fields in the same order in all maps.) You only need one copy of the symbolic map and you can use the same code to build the map.

CICS will select the physical map to use from the value coded in the ALTSUFFIX option of the TYPETERM resource definition for the terminal from which the transaction is being run. You also need to specify SCRNSZE(ALTERNATE) in the transaction PROFILE resource definition. See *CICS Resource Definition Guide* for information about the TYPETERM and PROFILE resource definitions.

You might use this technique to distinguish among standard terminals used for special purposes. For example, if an application were used by both English and French speakers, you could create two sets of physical maps, one with the constants in French and the other in English. You would assign each a suffix, and specify the English suffix as the ALTSUFFIX value in the definitions of the English terminals and the French suffix for French terminals. Transactions using the map would point to a PROFILE that specified alternate screen size. Then when you sent the map, BMS would pick the version with the suffix that matched the terminal (that is, in the appropriate language).

Another way to provide device dependent maps is to allow BMS to generate a suffix based on the terminal type, and select the physical map to use to match the terminal in the current execution when you issue SEND MAP or RECEIVE MAP.

Device dependent support: DDS

The BMS feature that does this is called “device dependent support” (DDS). DDS is an installation option that works as follows.

When you assemble your map sets, you specify the type of terminal the maps are for in the TERM option. This causes the assembler to store the physical map set under the MAPSET name suffixed by the character for that type of terminal⁵. When you issue SEND MAP or RECEIVE MAP with DDS active, BMS adds a 1-character suffix to the name you supply in the MAPSET option. It chooses the suffix based on the definition of your terminal, and thus loads the physical map that corresponds to the terminal for any given execution.

5. You also can use JCL or the link-edit NAME statement to control the member name under which a map set is stored.

BMS defines the suffixes used for the common terminal types. A 3270 Model 2 with a screen size of 24 rows and 80 columns is assigned the letter 'M,' for example. The type is determined from the TYPETERM definition if it is one of the standard types shown in Table 39.

Table 39. Terminal codes for BMS

Code	Terminal or logical unit
A	CRLP (card reader input, line printer output) or TCAM terminal. Note: In CICS TS 3.1, local TCAM terminals are not supported. The only TCAM terminals supported are remote terminals connected to a pre-CICS TS 3.1 terminal-owning region by the DCB (not ACB) interface of TCAM.
B	Magnetic tape
C	Sequential disk
D	TWX Model 33/35
E	1050
F	2740-1, 2740-2 (without buffer receive)
G	2741
H	2740-2 (with buffer receive)
I	2770
J	2780
K	3780
L	3270-1 displays (40-character width)
M	3270-2 displays (80-character width), LU type 2s
N	3270-1 printers
O	3270-2 printers, LU type 3s
P	All interactive LUs, 3767/3770 Interpreter LU, 3790 full function LU, SCS printer LU
Q	2980 Models 1 and 2
R	2980 Model 4
U	3600 (3601) LU
V	3650 Host Conversational (3653) LU
W	3650 Interpreter LU
X	3650 Host Conversational (3270) LU
Y	3770 Batch LU, 3770 and 3790 batch data interchange LUs, LU type 4s
blank	3270-2 (default if TERM omitted)

An installation can also define additional terminal types, such as the miniature screen described above. The system programmer does this by assigning an identifier to the terminal type and specifying it in the ALTSUFFIX option of the TYPETERM definition for the terminals. When you create a map for such a terminal, you specify this identifier in the SUFFIX option, instead of using the TERM option. Transactions using the map must also point to a PROFILE that specifies alternate screen size, so that ALTSUFFIX is used.

With DDS, the rules BMS uses for selecting a physical map are:

- BMS adds the ALTSUFFIX value in the terminal definition to your map set name, provided that definition specifies both ALTSUFFIX and ALTSCREEN, and

provided that the screen size for the transaction is the alternate size (either because the transaction PROFILE calls for alternate size, or because the default and alternate sizes are the same).

- If these conditions are not met, or if BMS cannot find a map with that suffix, it attempts to find one with the suffix that corresponds to the terminal type in the terminal definition.
- If BMS cannot find that map either, it looks for one with no suffix. (A blank suffix indicates an all-purpose map, suitable for any terminal that might use it.)

Without DDS, BMS always looks first (and only) for an unsuffixed map.

Device-dependent support is an installation option for BMS, set by the system programmer in the system initialization table. Be sure that it is included in your system before taking advantage of it; you should know whether it is present, even if you are supporting only one device type.

With DDS in the system, there is an efficiency advantage in creating suffixed map sets, even if you are supporting only one device type, because you prevent BMS from attempting to load a map set that does not exist before defaulting to the universal one (the blank suffix).

Without DDS, on the other hand, it is unnecessary to suffix your maps, because BMS looks for the universal suffix (a blank) and fails to locate suffixed maps.

Finding out about your terminal

Because of the overall design of BMS, and device-dependent support in particular, you generally do not need to know much about your terminal to format for it. However, if you need to know the characteristics of your principal facility, you can use the ASSIGN and INQUIRE commands. You can tell, for example, whether your terminal supports a particular extended attribute, what national language is in use, screen size and so on. This type of information applies whether you are using BMS or terminal control to communicate with your terminal. You need it more often for terminal control, and so we describe the options that apply in that chapter, in “Finding out about your terminal” on page 375.

There are also ASSIGN options specific to BMS, but you need them most often when you use the ACCUM option, and so we describe them later, in “ASSIGN options for cumulative processing” on page 517.

Chapter 33. Sending BMS mapped output

When you have assembled your symbolic map set, you are ready to code. We have explained by example how you get data from an application program to a map. We discuss that process in greater detail now, describing all the steps that must be performed, and telling you more about the options you have.

You must perform the following steps to produce mapped output:

1. Acquire storage in which to build the map.
2. Copy the symbolic map set so that it defines the structure of this storage.
3. Initialize it.
4. Move the output data into the map structure.
5. Set the field attributes.
6. Write the map to the screen with a SEND MAP command, adding any device control information required.

This chapter describes:

- “Acquiring and defining storage for the maps”
- “Initializing the output map” on page 478
- “Moving the variable data to the map” on page 479
- “Setting the display characteristics” on page 479

Acquiring and defining storage for the maps

The first step in creating mapped output is to provide storage in which to arrange the variable map data that your program passes to BMS. If you place the map structure in working storage, CICS does the allocation for you. (CICS allocates a private copy of working storage for each execution of a program, so that data from one task does not get confused with that from another, as explained in “Program storage” on page 158.) To use working storage, copy the symbolic map set there with the language statement provided for the purpose:

COPY	in COBOL and assembler
%INCLUDE	in PL/I
#include	in C and C++

Working storage is the WORKING-STORAGE SECTION in COBOL, automatic storage in PL/I, C, C++, and DFHEISTG in a CICS assembler program. For example:

```
WORKING-STORAGE SECTION.  
...  
01  COPY QCKSET.  
...
```

Alternatively, you can obtain and release map set storage as you need it, using CICS GETMAIN commands. (GETMAIN is discussed in Chapter 46, “Storage control,” on page 577.) In this case you copy the map into storage addressed by a pointer variable (the LINKAGE SECTION in COBOL, based storage in PL/I, C, and C++, a DSECT in assembler). On return from the GETMAIN, you use the address returned in the SET option to associate the storage with the data structure, according to the facilities of the programming language.

We used working storage in the example back on page Figure 105 on page 457, but we could have used a GETMAIN. If we had, the code we just showed you would change to:

```
LINKAGE SECTION.  
...  
01 COPY QCKSET.  
...  
PROCEDURE DIVISION.  
...  
MOVE LENGTH OF QCKMAPO TO LL.  
EXEC CICS GETMAIN SET(ADDRESS OF QCKMAPO)  
      LENGTH(LL) END-EXEC.  
...
```

The length you need on your GETMAIN command is the length of the variable whose name is the map name suffixed by the letter “O”. In COBOL, PL/I, C, and C++, you can use language facilities to determine this length, as in the example above. In assembler, it is defined in an EQUate statement whose label is the map name suffixed by “L”.

BASE and STORAGE options

Two options on the DFHMSD map set definition macro affect how storage for maps is defined: BASE and STORAGE=AUTO (the STORAGE option always has the value AUTO). You can use either one or neither, so there are *three* possibilities. If you specify neither for a map set containing several maps, the symbolic structures for the maps are defined so that they overlay one another. If you specify STORAGE=AUTO, they do not; each occupies separate space. Thus STORAGE=AUTO requires more storage.

However, when you use maps that overlay one another in a single program, you must use them serially or compensate for the reuse of storage by programming. Unless storage is a major issue, STORAGE=AUTO simplifies programming and reduces the risk of error.

In PL/I, C, and C++, STORAGE=AUTO has the additional effect of defining the map as automatic storage (storage that CICS allocates); the absence of STORAGE=AUTO causes these compilers to assume based storage, for which you generally incur the overhead of an additional GETMAIN. BMS assigns the name BMSMAPBR to the associated pointer variable, unless you specify another name with the BASE option.

The third possibility, BASE, lets you use the same storage for all the maps in multiple map sets. Its effect varies slightly with the programming language, but essentially, all the maps in map sets with the same BASE value overlay one another. In COBOL, BASE=xxxx causes the 01 levels (that is, each individual map) to contain a REDEFINES xxxx clause. In PL/I, C, and C++, it designates each map as storage based on the pointer variable xxxx. BASE cannot be used when the programming language is assembler.

Initializing the output map

Before you start building your output, make sure that the map storage is initialized to nulls, so that data left there by a previous process is not used inadvertently. If you have read input data using this same map, or one that overlays it, you need to ensure that you have processed or saved this data first. The relationship between

input and output maps is discussed in “The symbolic input map” on page 495, and using the same map you used for input is discussed in “Sending mapped output after mapped input” on page 503.

You initialize by moving nulls (X'00') into the structure. The symbolic map structures are defined so that you can refer to the whole map area by the name of the map suffixed by the letter *O*. You can see this in Figure 104 on page 456, and, in fact, the statement:

```
MOVE LOW-VALUES TO QCKMAPO.
```

would clear the area in which we built the map in the “quick check” example. If you are using the map for both input and output, it may be easier to clear the map one field at a time, as you edit the input (see “Handling input errors” on page 501).

When you obtain map storage with a CICS GETMAIN instruction, another way to initialize is to use the INITIMG option.

Moving the variable data to the map

Having obtained storage for your map, established the relationship of the map structure to the storage, and initialized, you are finally ready to create your output. There are two parts to it: the data itself and its display attributes. We tell you about the data first and get to the attributes right after.

In the usual case, an output display consists of some constant or default data (provided by the physical map) and some variable data (provided by the program). For each field that you want to supply by program, you move the data into the field in the symbolic map whose name is the name assigned in the map suffixed by the letter *O*. See the code on page “A BMS output example” on page 457 for an example.

If you do not supply a value for a field (that is, you leave it null, as initialized), BMS ordinarily uses the initial value assigned in the map, if any. Constants (that is, fields without names) also get the initial values specified in the map. However, the DATAONLY and MAPONLY options on the SEND MAP command modify the way in which program and map data are merged; we explain these options in “Merging the symbolic and physical maps” on page 484 and summarize the exact rules in “Building the output screen” on page 485.

Setting the display characteristics

Display attributes are the second component of the output data. (See “3270 field attributes” on page 392 for information about attributes.) In the “quick check” example on page “A BMS output example” on page 457, we show how 3270 field attributes for a map field are defined with the ATTRB option, and how BMS generates the “A” subfield to let you override the map value by program if you name the field.

BMS always provides the A subfield, because all 3270 devices support field attributes. Many 3270s also have some of the extended attributes shown in Table 40 on page 480. BMS supports each of these attributes individually in much the same way that it does field attributes collectively. You can assign attribute values in your DFHMDf field definitions, and, if you name the field, BMS generates a subfield in the symbolic map, so that you can override the map-assigned value in your program. There is a separate subfield for each type of extended attribute.

You can request subfields for the extended attributes by specifying the required attribute in the DSATTS option of DFHMDI or DFHMSD. You must also include the list of extended attributes in the the MAPATTS option (even if these attribute types do not appear in any DFHMDF macro).

Table 40. BMS attribute types. The columns show the types of attributes, the name of the associated MAPATTS and DSATTS value, and the suffix of the associated subfields in the symbolic map.

Attribute type	MAPATTS, DSATTS value	Subfield suffix
Field attributes	None (default)	A
Color	COLOR	C
Highlighting	HILIGHT	H
Outlining	OUTLINE	U
Background transparency	TRANSP	T
Validation	VALIDN	V
Double-byte character capability	SOSI	M
Programmed symbols	PS	P

Note: If you use programmed symbols, you need to ensure that a suitable symbol set has been sent to the device first, unless you choose one that is permanently loaded in the device. You can use a terminal control SEND command to do this (see “Using data transmission commands” on page 370). The *IBM 3270 Information Display System Data Stream Programmer's Reference* manual describes what to send.

The types of attributes that apply depend on the features of your principal facility at the time of execution. If you specify a value for an attribute that the terminal does not possess, BMS ignores it. If you are supporting different terminal types, however, you may need to use different techniques to get the same visual clarity. You can find out what kind of terminal you are using with the ASSIGN and INQUIRE commands, explained in “Finding out about your terminal” on page 375. There are also provisions in BMS for keeping your program independent of the terminal type; see “Device-dependent maps” on page 472.

Changing the attributes

Here is an example of how this works. Suppose that the terminals in our “quick check” application have color and highlighting capabilities. We might decide to show the maximum charge allowed in a different color from the rest of the screen, because this field is of most interest to the clerk. We might also make the warning message red, because when it appears at all, it is important for the clerk to notice it. And when we really want to get the clerk’s attention, because the card is stolen, we could change the attributes in the program to make the message blink. To add these features, we need to change our map definition as follows:

```
QCKMAP  DFHMDI SIZE=(24,80),...,X
        MAPATTS=(COLOR,HILIGHT),COLOR=GREEN,HILIGHT=OFF,DSATTS=HILIGHT
```

The MAPATTS option tells BMS that we specify color and highlighting in the map (or in the program, because any attribute listed in DSATTS must be included in MAPATTS as well). The COLOR and HILIGHT values indicate that fields with no color assigned should be green and that highlighting should be off if not specified.


```
# The only field definitions that we need to change are the ones that are not green or
# are highlighted:
# CHG      DFHMDF POS=(5,13),LENGTH=8,ATTRB=(ASKIP,NORM),PICOUT='$,$$0.00',      X
#          COLOR=NEUTRAL
# MSG      DFHMDF LENGTH=20,POS=(7,1),ATTRB=(ASKIP,NORM),COLOR=RED
#
# Specifying COLOR=NEUTRAL means that on a terminal, the field is displayed in
# white.
```

The DSATTS option tells BMS that we want to alter the highlighting of some fields at execution time, and therefore it should produce “H”-suffix subfields in the symbolic map to let us do that. Each named field gets the extra subfield; the message field, for example, expands from the current three lines in Figure 104 on page 456 to:

```
02 FILLER PICTURE X(2).
02 MSGH    PICTURE X.
02 MSGA    PICTURE X.
02 MSGO    PIC X(30).
```

The program statement we need to produce the blinking is:

```
MOVE DFHBLINK to MSGH.
```

In general, BMS takes attribute values from the program if you supply them and from the map if you do not (that is, if you leave the program value null, as initialized). However, the MAPONLY and DATAONLY options on the SEND MAP command affect attribute values as well as field data, as explained in “Where the values come from” on page 486.

Attribute value definitions: DFHBMSCA

The 1-byte values required to set attribute values are bit combinations defined by 3270 hardware. They are hard to remember and, in some languages, clumsy to express. To solve this problem, CICS provides source code that you can copy into your program. The code, named DFHBMSCA, defines all the commonly used values for all attributes and assigns meaningful names to each combination. DFHBLINK in the line of code above is an example. To define DFHBLINK, we simply copy DFHBMSCA into our working storage, thus:

```
WORKING-STORAGE SECTION.
...
01 COPY DFHBMSCA.
```

There is a separate version of DFHBMSCA for each programming language, but the value names are the same in all versions. If you need an attribute combination not included in DFHBMSCA, you can determine the value by referring to the *IBM 3270 Information Display System Data Stream Programmer's Reference*; if you make frequent use of the value, you may want to modify DFHBMSCA to include it.

Note: In assembler language only, the values are defined with EQUates, so you use MVI rather than MVC instructions.

Chapter 34. Using the SEND MAP command

The SEND MAP command tells BMS:

- Which map to use (MAP option), and where to find that map (the MAPSET option)
- Where to find the variable data for the map (FROM option) and how to merge it with the values from the map (MAPONLY and DATAONLY)
- Which device controls to include in the data stream, and other control options
- Where to put the cursor, if you want to override the position in the map definition (the CURSOR option)
- Whether the message is complete or is built cumulatively (the ACCUM option)
- What to do with the formatted output (TERMINAL, SET and PAGING options)

The MAP and MAPSET options are self-explanatory, and we cover most of the rest as we describe the programming steps that precede a simple SEND MAP . The last two topics require a knowledge of BMS logical message facilities, which we describe in “Output disposition options: TERMINAL, SET, and PAGING” on page 489.

Until we get to that point, we assume the defaults: that each SEND MAP creates one message, and we are sending that message to our own terminal. The *CICS Application Programming Reference* manual describes this command in more detail.

This chapter describes:

- “SEND MAP control options”
- “Merging the symbolic and physical maps” on page 484
- “Building the output screen” on page 485
- “Positioning the cursor” on page 488
- “Sending invalid data and other errors” on page 489
- “Output disposition options: TERMINAL, SET, and PAGING” on page 489

SEND MAP control options

There are many control options for the BMS SEND commands. Some apply only to particular devices or special features of BMS, and we defer describing these until we get to the associated device support or feature. The following device control options, however, apply generally:

- **ERASE**, **ERASEAUP**, and **FRSET** all modify the contents of the device buffer, if the terminal has one, before writing your output into it. ERASE sets the entire buffer to nulls (X'00'). If the terminal has the alternate screen size feature, ERASE also sets the buffer size. Therefore, the first SEND MAP in a task normally specifies the ERASE option, both to clear the buffer and to select the buffer size. (See “3270 write commands” on page 390 for more information about alternate screen size.)

ERASEAUP (erase all unprotected fields) sets the contents of all fields in the buffer that are unprotected (that is, fields which the operator can change) to nulls. This is useful for data entry, as we explain in “DATAONLY option” on page 485.

FRSET (field reset) turns off the modified data tag of all fields in the buffer (“Saving the good input” on page 502 and “Modification” on page 393 explain more about this option).

- **FREEKB** (free keyboard) unlocks the keyboard when the output is sent to the terminal. You usually want to do this on a display terminal.

- **ALARM** sounds the audible alarm, if the terminal has one.
- **FORMFEED, PRINT, L40, L64, L80,** and **HONEOM** are specific to printing and are explained in “CICS 3270 printer options” on page 409. **NLEOM** also is used mainly in printing, and is explained in the same section. NLEOM requires **standard BMS**.

Some of these options can also be specified in the map itself, in particular, the options that are expressed in the 3270 write control character and coded in the CTRL option of the DFHMDI or DFHMSD macros: PRINT, FREEKB, ALARM, FRSET, L40, L64, L80, HONEOM.

Note: CTRL options are always treated as a group, so if you include any of them on your SEND MAP command, BMS ignores the values for *all* of them in your map definition and uses only those in the command. As we noted earlier, you can also send device control options separate from your map data, using a SEND CONTROL command. You can use any option on SEND CONTROL that you can use on SEND MAP, except those that relate expressly to data, such as NLEOM.

Other BMS SEND options: WAIT and LAST

When a task writes to a terminal with a BMS or terminal control SEND command CICS normally schedules the transmission and then makes the task ready for execution again. Actual transmission occurs somewhat later, depending on terminal type, access method and other activity in the system. If you want to ensure that transmission is complete before your program regains control, use the WAIT option.

WAIT can increase response time slightly, because it prevents overlap between processing and output transmission for a task. (Overlap occurs only until a subsequent SEND, RECEIVE, or end of task, however, because CICS finishes one terminal operation completely before starting another.)

You can improve response time slightly for some terminals by using the LAST option. LAST indicates that the output you are sending is the last output for the task. This knowledge allows CICS to combine transmission of the data with the VTAM end-of-bracket flow that occurs at end of task.

Merging the symbolic and physical maps

So far, we have assumed that every display consists of some constant data (provided by the physical map) and some variable data (provided by the program and structured according to the symbolic map). Sometimes, however, one or more of these components is missing.

MAPONLY option

For example, a menu map may not need any data supplied by program. In such a case, you code the MAPONLY option in place of the FROM option on your SEND MAP command. BMS then takes all the information from the physical map, sending the initial values for both the constant (unnamed) and named fields. You do not need to copy the symbolic map set into a program that always sends the map with MAPONLY, and, in fact, you can skip the TYPE=DSECT map set assembly if all programs use all the maps in the set in this way.

MAPONLY is also the way you get an input-only map to the screen.

DATAONLY option

The opposite situation is also possible: the program can supply all the data and not need any constant or default values from the map. This happens on the second and subsequent displays of a map in many situations: data entry applications, inquiry applications where the operator browses through a series of records displayed in identical format, and screens which are redisplayed after detection of an error in the input.

BMS takes advantage of this situation if you indicate it with the DATAONLY option. You still need to tell BMS which map and map set you are using for positioning information, but BMS sends only those fields which have non-null attribute or data values in the symbolic map. Other fields and attribute values are left unchanged.

The SEND CONTROL command

There are also occasions when you do not need to send data at all, but you do need to send device controls. For example, you might need to erase the screen or sound the alarm. You do this with a SEND CONTROL command listing the options you need,

Consider a program in a data entry application. When first initiated, it displays the data entry map to format the screen with the input fields, the associated labels, screen headings and instructions. This first SEND MAP command specifies MAPONLY, because the program sends no variable data. Thereafter, the program accepts one set of data input. If the input is correct, the program files it and requests another. It still does not need to send any variable data. What it needs to do is to erase the input from the screen and unlock the keyboard, to signal the operator to enter the next record.

```
EXEC CICS SEND CONTROL ERASEAUP FREEKB END-EXEC
```

does this. (See “SEND MAP control options” on page 483 for a description of these and other device control options.)

If there are errors, the program does need to send variable data, to tell the operator how to fix the problem. This one changes the attributes of the fields in error to highlight them and sends a message in a field provided for the purpose. Here, our program uses the DATAONLY option, because the map is already on the screen. (We tell you more about handling input errors in “Handling input errors” on page 501.)

You should use MAPONLY, DATAONLY, and SEND CONTROL when they apply, especially when response time is critical, as it is in a data entry situation. MAPONLY saves path length, DATAONLY reduces the length of the outbound data stream, and SEND CONTROL does both.

Building the output screen

The interaction of physical map definition options, SEND MAP options, program data and merge options is sufficiently complex that a summary of the rules for determining what appears on the screen after a SEND MAP is in order.

The contents of the screen (buffer) are determined by:

- What was there before your SEND MAP command
- The fields (field attributes, extended attributes, and display data) that get sent from your SEND MAP command

- Where the several values for these field elements come from

We discuss the possibilities in that order.

What you start with

The first thing that happens on a SEND MAP command is that the entire screen (buffer) is cleared to nulls if the ERASE option is present, regardless of the size or origin of your map. On terminals that have the alternate screen size feature, the screen size is set as well, as explained in “3270 write commands” on page 390. The screen is in unformatted state, with no fields defined and no display data. If ERASEAUP is present, all of the unprotected fields on the screen are erased, but the field structure and attributes of all fields and the contents of protected fields are unchanged.

ERASE and ERASEAUP are honored *before* your SEND MAP data is loaded into the buffer. If neither of these options appears on the SEND MAP, the screen buffer starts out as it was left after the previous write operation, modified by whatever the operator did. In general, the positions of the fields (that is, of the attributes bytes) and their attributes are unchanged, but the data content of unprotected fields may be different. Furthermore, if the operator used the CLEAR key, the whole buffer is cleared to nulls and the screen is in unformatted state, just as if you had included the ERASE option.

What is sent

Secondly, BMS changes only those positions in the buffer within the confines of your map. Outside that area, the contents of the buffer are unchanged, although it is possible for areas outside your map to change in appearance, as explained in “Outside the map” on page 487.

Within the map area, what is sent depends on whether the DATAONLY option is present. In the usual case, where it is not present, BMS sends every component (field attributes, extended attributes, display data) of every field in your map. This creates a field at the position specified in the POS operand and overlays the number of bytes specified in the LENGTH field from POS. Buffer positions between the end of the display data and the next attributes byte (POS value) are left unchanged. (There may or may not be fields (attributes bytes) in these intervening spaces if you did not ERASE after the last write operation that used a different map.)

The values for these field elements come from the program, the map or defaults, as explained in the next section.

If DATAONLY is present, on the other hand, BMS sends only those fields, and only those components for them, that the program provides. Other screen data is unchanged.

Where the values come from

The values that determine screen contents may come from four sources:

- Program
- Map
- Hardware defaults
- Previous screen contents

BMS considers each component of each map field separately, and takes the value from the program, provided:

- The MAPONLY option has not been used.
- The field has a name in the map, so that the symbolic output map contains the corresponding set of subfields from which to get the data. The field attributes value comes from the program subfield whose name is the map field name suffixed by A. The display data comes from the subfield of the same name suffixed by O, and the extended attribute values come from the same-named subfields suffixed by the letter that identifies the attribute (see Table 40 on page 480). In the case of the extended attributes, the attribute must also appear among DSATTS in order for the symbolic map to contain the corresponding subfield.
- A value is present. The definition of “present” varies slightly with the field component:
 - For field attributes bytes, the value must not be null (X'00') or one of the values that can be left over from an input operation (X'80', X'02', or X'82').
 - For extended attribute bytes, the value must not be null.

Note: BMS sends only those extended attribute values that the terminal is defined as supporting. Values for other extended attributes are omitted from the final data stream.

- For display data, the first character of the data must not be null.

If any of these conditions is not met, the next step depends on whether DATAONLY is present. With DATAONLY, BMS stops the process here and sends only the data it got from the program. BMS does this in such a way that components not changed by program are not changed on the screen. In particular, extended attributes values are not changed unless you specify a new value or ask for the hardware default. (A value of X'FF' requests the hardware default for all extended attributes except background transparency, for which you specify X'F0' to get the hardware default.)

Without DATAONLY, if one of the conditions above is not met, BMS takes the data from the map, as follows:

- For field attributes, it takes the value in the ATTRB option for the field. If none is present, BMS assumes an ATTRB value of (ASKIP,NORM).
- For extended attributes, BMS takes the value from:
 - The corresponding option in the DFHMDF field definition
 - If it is not specified there, the value is taken from the corresponding option in the DFHMDI map definition
 - If it is not there either, the value is taken from the corresponding option in the DFHMSD map set definition

(If no value is specified anywhere, BMS does not send one, and this causes the 3270 to use its hardware default value.)

- For display data, from the initial value in the map (the INITIAL, XINIT, or GINIT option). If there is no initial value, the field is set to nulls.

Outside the map

We have assumed, so far, that your map is the same size as your screen or printer page. It need not be. Your application may use only a part of the screen area, or you may want to build your output incrementally, or both.

BMS logical messages allow you to build a screen from several maps, sending it with a single terminal write. You use the ACCUM option to do this, which we cover in Chapter 36, “BMS logical messages,” on page 507. Even without using ACCUM, you can build a screen from several maps if the terminal is a 3270-like device with

a buffer. You do this with multiple SEND MAP commands written to different areas of the screen (buffer), not erasing after the first command. Each SEND MAP causes output and may produce a momentary “blink” at a display device. For this reason, and to eliminate the path length of extra I/O, you may prefer to use logical messages for such composite screens.

Outside the map just sent, the contents of the buffer are unchanged, except for the effects of ERASE and ERASEAUP cited earlier. In general, this means that the corresponding areas of the screen are unchanged. However, a screen position outside the map may get its attributes from a field within the map. This occurs if you do not define a field (using a different map) beyond the boundary of the map and before the position in question. If you change the attributes of the field inside your map governing this position outside, the appearance of the position may change, even though the contents do not.

Using GDDM and BMS

One use of the buffer overlay technique we just described is the creation of screens containing a mixture of BMS and Graphical Data Display Manager (GDDM) output. You generally write the BMS output first, followed by the GDDM. You can leave space in the BMS map for the GDDM output, or you can create a “graphic hole” in any display by writing a map with no fields in it to the position where you want the hole. Such a map is called a “null map,” and its size (height and width) correspond to the size of the hole.

If you use GDDM to combine graphics with BMS output, you need to include a GDDM PSRSRV call to prevent GDDM from corrupting programmed symbol sets that BMS may be using.

Positioning the cursor

Usually, you set the initial position for the cursor in the map definition by including “insert cursor” (IC) in the ATTRB values of the field where you want it. (Cursor position is not important for the output-only maps we have been discussing, but it becomes important as soon as you use a map for input too.)

The CURSOR option on the SEND MAP command allows you to override this specification, if necessary, when the map is displayed. If you specify CURSOR(value), BMS places the cursor in that absolute position on the screen. Counting starts in the first row and column (the zero position), and proceeds across the rows. Thus, to place the cursor in the fourth column of the third row of an 80-column display, you code CURSOR(163).

Specifying CURSOR without a value signals BMS that you want “symbolic cursor positioning”. You do this by setting the length subfield of the field where you want the cursor to minus one (-1). Length subfields are not defined on output-only maps, so you must define your map as INOUT to use symbolic cursor positioning. (We tell you about length subfields in “Formatted screen input” on page 497, and about INOUT maps in Chapter 35, “Receiving mapped data,” on page 493.) If you mark more than one field in this way, BMS uses the first one it finds.

Symbolic cursor positioning is particularly useful for input-output maps when the terminal operator enters incorrect data. If you validate the fields, setting the length of any in error to -1, BMS places the cursor under the first error when you redisplay. “Processing the mapped input” on page 500 shows this technique.

You can position the cursor with a SEND CONTROL command also, but only by specifying an absolute value for CURSOR; if you omit CURSOR on SEND CONTROL, the cursor is not moved.

Sending invalid data and other errors

The exceptional conditions that can occur on SEND MAP and SEND CONTROL commands are listed with the descriptions of these commands in the *CICS Application Programming Reference* manual. Most of them apply only to the advanced BMS options: logical messages, partitions, and special devices.

However, it is also possible to send invalid data to a terminal. BMS does not check the validity of attribute and data values in the symbolic map, although it does not attempt to send an extended attribute, like color, to a terminal not defined to support that attribute.

The effects of invalid data depend on both the particular terminal and the nature of the incorrect data. Sometimes invalid data can be interpreted as a control sequence, so that the device accepts the data but produces the wrong output; sometimes the screen displays an error indicator; and sometimes an ATNIabend occurs. The point at which your task is notified of an ATNI depends on whether or not you specified the WAIT option (see “Other BMS SEND options: WAIT and LAST” on page 484).

Output disposition options: TERMINAL, SET, and PAGING

The only disposition option we have described up to this point is TERMINAL, which sends the output to the principal facility of your task. TERMINAL is the default value that you get if you do not specify another disposition. There are, however, two other possibilities:

1. BMS can return the formatted output stream to the task rather than sending it to the terminal. You use the SET disposition option to request this. You might do so to defer transmission or to modify the data stream to meet special requirements. “Acquiring and defining storage for the maps” on page 477 explains how and when to use SET.
2. You can ask BMS to store and manage your output in CICS temporary storage for subsequent delivery to your terminal. This option, PAGING, implies that your message may contain more than one screen or page, and is particularly useful when you want to send a message to a display terminal that exceeds its screen capacity. BMS saves the entire message in temporary storage until you indicate that it is complete. Then it provides facilities for the operator to page through the output at the terminal. You can use PAGING for printers as well as displays, although you do not need the operator controls, and sometimes TERMINAL is just as satisfactory.

When you use PAGING, the output still goes to your principal facility, though indirectly, as just described. Full BMS also provides a feature, routing, that lets you send your message to another terminal, or several, in place of or in addition to your own. We tell you about routing in Chapter 39, “Message routing,” on page 527, after we cover the prerequisites.

Note: Both PAGING and SET and related options require **full** BMS. TERMINAL is the only disposition available in minimum and standard BMS.

Using SET

When you specify a disposition of SET for a BMS message, BMS formats your output and returns it in the form of a device-dependent data stream. No terminal I/O occurs, although the returned data stream usually is sent to a terminal subsequently.

There are several reasons for asking BMS to format a data stream without sending it. You might want to do any of the following:

- Edit the data stream to meet the requirements of a device with special features or restrictions not explicitly supported by CICS.
- Compress the data stream, based on standard 3270 features or special device characteristics.
- Forward the data stream to a terminal not connected directly to CICS. For example, you might want to pass data to a 3270 terminal attached to a system connected to CICS by an APPC link. You can format the data with SET and send the resulting pages to a partner program across the link. If the terminal is of a different type from your principal facility, you can define a dummy terminal of the appropriate type and then ROUTE to that terminal with SET to get the proper formatting, as explained in “Routing with SET” on page 535.

BMS returns formatted output by setting the pointer variable named in the SET option to the address of a page *list*. This list consists of one or more 4-byte entries in the following format, each corresponding to one page of output.

Table 41. Page list entry format

Bytes	Contents
0	Terminal type (see Table 39 on page 474)
1-3	Address of TIOA containing the formatted page of output

An entry containing -1 (X'FF') in the terminal type signals the end of the page list. Notice that the addresses in this list are only 24 bits long. If your program uses 31-bit addressing, you must expand a 24-bit address to a full word by preceding it with binary zeros before using it as an address.

Each TIOA (terminal input-output area) is in the standard format for these areas:

Table 42. TIOA format

Field name	Position	Length	Contents
TIOASAA	0	8	CICS storage accounting information (8 bytes)
TIOATDL	8	2	Length of field TIOADBA in halfword binary format
(unnamed)	10	2	Reserved field
TIOADBA	12	TIOATDL	Formatted output page
(unnamed)	TIOATDL + 12	4	Page control area, required for the SEND TEXT MAPPED command (if used)

The reason that BMS uses a list to return pages is that some BMS commands produce multiple pages. SEND MAP does not, but SEND TEXT can. Furthermore, if you have established a routing environment, BMS builds a separate logical message for each of the terminal types among your destinations, and you may get pages for several different terminal types from a single BMS command. The

terminal type tells you to which message a page belongs. (Pages for a given type are always presented in order.) If you are not routing, the terminal type is always that of your principal facility.

If you are not using the ACCUM option, pages are available on return from the BMS command that creates them. With ACCUM, however, BMS waits until the available space on the page is used. BMS turns on the RETPAGE condition to signal your program that pages are ready. You can detect RETPAGE with a HANDLE CONDITION command or by testing the response from the BMS command (in EIBRESP or the value returned in the RESP option).

You must capture the information in the page list whenever BMS returns one, because BMS reuses the list. You need save only the addresses of the pages, not the contents. BMS does not reuse the pages themselves, and, in fact, moves the storage for them from its control to that of your task. This allows you to free the storage for a page when you are through with it. If you do this, the DATA or DATAPOINTER option in your FREEMAIN command should point to the TIOATDL field, not to TIOASAA.

Chapter 35. Receiving mapped data

Formatted screens are as important for input as for output. Data entry applications are an obvious example, but most other applications also use formatted input, at least in part. On input, BMS does for you approximately the reverse of what it does on output: it removes device control characters from the data stream and moves the input fields into a data structure, so that you can address them by name.

Maps can be used exclusively for input, exclusively for output (the case we have already covered), or for both. Input-only maps are relatively rare, and we cover them as a special case of an input-output map, pointing out differences where they occur.

This chapter describes:

- “An input-output example”
- “Programming mapped input” on page 496
- “Using the RECEIVE MAP command” on page 496
- “Getting storage for mapped input” on page 497
- “Formatted screen input” on page 497
- “Handling input errors” on page 501
- “Finding the cursor” on page 500
- “Processing the mapped input” on page 500
- “Handling input errors” on page 501
- “Sending mapped output after mapped input” on page 503
- “MAPFAIL and other exceptional conditions” on page 504
- “Formatting other input” on page 505

An input-output example

Before we explain the details of the input structure, let us re-examine the “quick check” example. Suppose that it is against our policy to let a customer charge up to the limit over and over again between the nightly runs when new charges are posted to the accounts. We want a new transaction that augments “quick check” processing by keeping a running total for the day.

In addition, we want to use the same screen for both input and output, so that there is only one screen entry per customer. In the new transaction, “quick update,” the clerk enters both the account number and the charge at the same time. The normal response is:

When we reject a transaction, we leave the input information on the screen, so that

QUP Quick Account Update

Current charge okay; enter next

Account: _____

Charge: \$ _____

Figure 110. Normal “quick update” response

the clerk can see what was entered along with the description of the problem:

QUP	Quick Account Update
Charge exceeds maximum; do not approve	
Account:	482554
Charge:	\$ 1000.00

Figure 111. “Quick update” error response

(Here again, we are oversimplifying to keep our maps short for ease of explanation.)

The map definition we need for this exercise is:

You can see that the map field definitions for this input-output map are very similar

```

QUPSET  DFHMSD TYPE=MAP,STORAGE=AUTO,MODE=INOUT,LANG=COBOL,TERM=3270-2
QUPMAP  DFHMDI SIZE=(24,80),LINE=1,COLUMN=1,CTRL=FREEKB
        DFHMDF POS=(1,1),LENGTH=3,ATTRB=(ASKIP,BRT),INITIAL='QUP'
        DFHMDF POS=(1,26),LENGTH=20,ATTRB=(ASKIP,NORM),                X
        INITIAL='Quick Account Update'
MSG      DFHMDF LENGTH=40,POS=(3,1),ATTRB=(ASKIP,NORM)
        DFHMDF POS=(5,1),LENGTH=8,ATTRB=(ASKIP,NORM),                X
        INITIAL='Account:'
ACCTNO   DFHMDF POS=(5,14),LENGTH=6,ATTRB=(UNPROT,NUM,IC)
        DFHMDF POS=(5,21),LENGTH=1,ATTRB=(ASKIP),INITIAL=' '
        DFHMDF POS=(6,1),LENGTH=7,ATTRB=(ASKIP,NORM),INITIAL='Charge:'
CHG       DFHMDF POS=(6,13),ATTRB=(UNPROT,NORM),PICIN='$$$$0.00'
        DFHMDF POS=(6,21),LENGTH=1,ATTRB=(ASKIP),INITIAL=' '
        DFHMSD TYPE=FINAL

```

Figure 112. Map definition for input-output map

to those for the output-only “quick check” map, if we allow for changes to the content of the screen. The differences to note are:

- The MODE option in the DFHMSD map set definition is INOUT, indicating that the maps in this map set are used for both input and output. INOUT causes BMS to generate a symbolic structure for input as well as for output for every map in the map set. If this had been an input-only map, we would have said MODE=IN, and BMS would have generated only the input structures.
- We put names on the fields from which we want input (ACCTNO and CHG) as well as those to which we send output (MSG). As in an output-only map, we avoid naming constant fields to save space in the symbolic map.
- The input fields, ACCTNO and CHG, are unprotected (UNPROT), to allow the operator to key data into them.
- IC (insert cursor) is specified for ACCTNO. It positions the cursor at the start of the account number field when the map is first displayed, ready for the first item that the operator has to enter. (You can override this placement when you send the map; IC just provides the default position.)
- Just after the ACCTNO field, there is a constant field consisting of a single blank, and a similar one after the CHG field. These are called “stopper” fields. Normally, they are placed after each input field that is not followed immediately by some other field. They prevent the operator from keying data beyond the space you provided, into an unused area of the screen.

If you define the stopper field as “autoskip”, the cursor jumps to the next unprotected field after the operator has filled the preceding input field. This is convenient if most of the input fields are of fixed length, because the operator does not have to advance the cursor to get from field to field.

If you define the stopper field as “protected,” but not “autoskip,” the keyboard locks if the operator attempts to key beyond the end of the field. This choice may be preferable if most of the input fields are of variable length, where one usually

has to use the cursor advance key anyway, because it alerts the operator to the overflow immediately. Whichever you choose, you should use the same choice throughout the application if possible, so that the operator sees a consistent interface.

- The CHG field has the option PICIN. PICIN produces an edit mask in the symbolic map, useful for COBOL and PL/I, and implies the field length. See *CICS Application Programming Reference* manual for details on using PICIN.

Figure 113 shows the symbolic map set that results from this INOUT map definition. The second part of this structure, starting at QUPMAPO, is the symbolic *output*

```
01 QUPMAPI.
  02 FILLER PIC X(12).
  02 FILLER PICTURE X(2).
  02 MSGL COMP PIC S9(4).
  02 MSGF PICTURE X.
  02 FILLER REDEFINES MSGF.
    03 MSGA PICTURE X.
  02 MSGI PIC X(40).
  02 ACCTNOL COMP PIC S9(4).
  02 ACCTNOF PICTURE X.
  02 FILLER REDEFINES ACCTNOF.
    03 ACCTNOA PICTURE X.
  02 ACCTNOI PIC X(6).
  02 CHGL COMP PIC S9(4).
  02 CHGF PICTURE X.
  02 FILLER REDEFINES CHGF.
    03 CHGA PICTURE X.
  02 CHGI PIC X(7) PICIN '$,$$0.00'.
01 QUPMAPO REDEFINES QUPMAPI.
  02 FILLER PIC X(12).
  02 FILLER PICTURE X(3).
  02 MSGO PIC X(40).
  02 FILLER PICTURE X(3).
  02 ACCTNO PICTURE X(6).
  02 FILLER PICTURE X(3).
  02 CHGO PIC X.
```

Figure 113. Symbolic map for “quick update”

map—the structure required to send data back to the screen. Apart from the fields we redefined, it looks almost the same as the one you would have expected if we had specified MODE=OUT instead of MODE=INOUT. See Figure 102 on page 455 for a comparison. The main difference is that the field attributes (A) subfield appears to be missing, but we explain this in a moment.

The symbolic input map

The first part of the structure, under the label QUPMAPI, is new. This is the symbolic *input* map—the structure required for reading data from a screen formatted with map QUPMAP. For each named field in the map, it contains three subfields. As in the symbolic output map, each subfield has the same name as the map field, suffixed by a letter indicating its purpose. The suffixes and subfields related to input are:

- L** the length of the input in the map field.
- F** the flag byte, which indicates whether the operator erased the field and whether the cursor was left there.
- I** the input data itself.

The input and output structures are defined so that they overlay one another field by field. That is, the input (I) subfield for a given map field always occupies the same storage as the corresponding output (O) subfield. Similarly, the input flag (F) subfield overlays the output attributes (A) subfield. (For implementation reasons, the order of the subfield definitions varies somewhat among languages. In COBOL, the definition of the A subfield moves to the input structure in an INOUT map, but it still applies to output, just as it does in an output-only map. In assembler, the input and output subfield definitions are interleaved for each map field.)

BMS uses dummy fields to leave space in one part of the structure for subfields that do not occur in the other part. For example, there is always a 2-byte filler in the output map to correspond to the length (L) subfield in the input map, even in output-only maps. If there are output subfields for extended attributes, such as color or highlighting, BMS generates dummy fields in the input map to match them. You can see examples of these fields (FILLERS in COBOL) in both Figure 102 on page 455 and Figure 113 on page 495.

The correspondence of fields in the input and output map structures is very convenient for processes in which you use a map for input and then write back in the same format, as you do in data entry transactions or when you get erroneous input and have to request a correction from the operator.

Programming mapped input

The programming required for mapped input is similar to that for mapped output, except, of course, that the data is going in the opposite direction. You define your maps and assemble them first, as for mapped output. In the program or programs reading from the terminal, you:

1. Acquire the storage to which the symbolic map set corresponds.
2. Copy the symbolic map set to define the structure of this storage.
3. Format the input data with a RECEIVE MAP command.
4. Process the input.

We tell you more about these tasks and related topics in the paragraphs that follow, starting with the RECEIVE MAP command. We also develop the code for the “quick update” transaction.

If the transaction also calls for mapped output, as “quick update” and most other transactions do, you simply continue with the steps outlined before, in Chapter 33, “Sending BMS mapped output,” on page 477. Some considerations and shortcuts for mapped input are described in “Sending mapped output after mapped input” on page 503.

Using the RECEIVE MAP command

The RECEIVE MAP command causes BMS to format terminal input data and make it accessible to your application program. It tells BMS:

- Which map to use in formatting the input data stream—that is, what format is on the screen and what data structure the program expects (the MAP option)
- Where to find this map (MAPSET option)
- Where to get the input (TERMINAL or FROM option)
- Whether to suppress translation to upper case (ASIS option)
- Where to put the formatted input data (the INTO and SET options)

The MAP and MAPSET options together tell BMS which map to use, and they work exactly as they do on a SEND MAP command.

BMS gets the input data to format from the terminal associated with your task (its principal facility), unless you use the FROM option. FROM is an alternative to TERMINAL, the default, used in relatively unusual circumstances (see “Formatting other input” on page 505).

BMS also translates lower case input to upper case automatically in some cases; we explain how to control translation in “Upper case translation” on page 499.

You tell BMS where to put the formatted input with the INTO or SET option of RECEIVE MAP. For the full syntax of the RECEIVE MAP command, see *CICS Application Programming Reference manual* .

In addition to the data on the screen, the RECEIVE MAP command tells you where the operator left the cursor and what key caused transmission. This information becomes available in the EIB on completion of the RECEIVE MAP command. EIBAID identifies the transmit key (the “attention identifier” or AID), and EIBCURSR tells you where the cursor was left.

Getting storage for mapped input

When you issue a RECEIVE MAP command, BMS needs storage in which to build the input map structure. You can provide this space yourself, either in the working storage of your program or with a CICS GETMAIN. These are the same choices you have for allocating storage in which to build an output map, and you use them the same way (see “Acquiring and defining storage for the maps” on page 477 for details and examples). For either, you code the INTO option on your RECEIVE command, naming the variable into which the formatted input is to be placed. For our “quick update”, for example, the required command is:

```
EXEC CICS RECEIVE MAP('QUPMAP') MAPSET('QUPSET')
      INTO(QUPMAPI) END-EXEC.
```

Usually, the receiving variable is the area defined by the symbolic input map, to which BMS assigns the map name suffixed by the letter “I”, as shown above. You can specify some other variable if you wish, however.

For input operations, you have a third choice for acquiring storage. If you code the SET option, BMS acquires the storage for you at the time of the RECEIVE command and returns the address in the pointer variable named in the SET option. So we could have coded the RECEIVE MAP command in “quick update” like this:

```
LINKAGE SECTION.
...
01 QUPMAP COPY QUPMAP.
...
PROCEDURE DIVISION.
...
EXEC CICS RECEIVE MAP('QUPMAP') MAPSET('QUPSET')
      SET(ADDRESS OF QUPMAPI) END-EXEC.
...
```

Storage obtained in this way remains until task end unless you issue a FREEMAIN to release it (see Chapter 46, “Storage control,” on page 577).

Formatted screen input

As we noted earlier, we explain receiving input from a terminal in terms of 3270 devices. You should also read “Support for non-3270 terminals” on page 470 if you are writing for non-3270 terminals.

CICS normally reads a 3270 screen with a “read modified” command⁶. The data transmitted depends on what the operator did to cause transmission:

- The ENTER key or a PF key
- CLEAR, CNCL or a PA key (the “short read” keys)
- Field selection: cursor select, light pen detect or a trigger field

You can tell which event occurred, if you need to know; we explain how in “Using the attention identifier” on page 499. You can also find more detail on 3270 input operations in “Input from a 3270 terminal” on page 400.

The short read keys transmit only the attention identifier (the identity of the key itself). No field data comes in, and there is nothing to map. For this reason, short read keys can cause the MAPFAIL condition, as explained on page 504. Field selection features transmit field data, but in most cases not the same data as the ENTER and PF keys, which we describe in the paragraphs that follow. See Chapter 42, “Support for special hardware,” on page 549 for the exceptions if you plan to use these features.

Most applications are designed for transmission by the ENTER key or a PF key. When one of these is used to transmit, all of the fields on the screen that have been modified, and *only* those fields, are transmitted.

Modified data

As we explained in “Modification” on page 393, a 3270 screen field is considered modified only if the “modified data tag” (MDT), one of the bits in the field attributes byte, is on. The terminal hardware turns on this bit if the operator changes the field in any way—entering data, changing data already there, or erasing. You can also turn it on by program when you send the map, by including MDT among the ATTRB values for the field. You do this when you want the data in a particular field to be returned even if the operator does not change it.

You can tell whether there was input from a particular map field by looking at the corresponding length (L) subfield. If the length is zero, no data was read from that field. The associated input (I) subfield contains all nulls (X'00'), because BMS sets the entire input structure to nulls before it performs the input mapping operation. The length is zero either if the modified data tag is off (that is, the field was sent with the tag off and the operator did not change it) or if the operator erased the field. You can distinguish between these two situations, if you care, by inspecting the flag (F) subfield. It has the high-order bit on if the field contains nulls but the MDT is on (that is, the operator changed the field by erasing it). See “Finding the cursor” on page 500 for more information about the flag subfield.

If the length is nonzero, data was read from the field. Either the operator entered some, or changed what was there, or the field was sent with the MDT on. You may find the data itself in the corresponding input (I) subfield. The length subfield tells how many characters were sent. A 3270 terminal sends only non-null characters, so BMS knows how much data was keyed into the field. Character fields are filled out with blanks on the right and numeric fields are filled on the left with zeros unless you specify otherwise in the JUSTIFY option of the field definition. BMS assumes that a field contains character data unless you indicate that it is numeric with ATTRB=NUM. See the *CICS Application Programming Reference* manual for details of how these options work.

6. CICS provides an option, BUFFER, for the terminal control RECEIVE command, with which you can capture the entire contents of a 3270 screen. See “Reading from a 3270 terminal” on page 403 if you need to do this.

Upper case translation

CICS converts lower case input characters to upper case automatically under some circumstances. The definition of the terminal and the transaction together determine whether translation occurs. See the UCTRAN option of the PROFILE and the TYPETERM definitions in *CICS Resource Definition Guide* for how these specifications interact.

You can suppress this translation by using the ASIS option on your RECEIVE MAP command, *except* on the first RECEIVE in a task initiated by terminal input. (The first RECEIVE may be either a RECEIVE MAP (without FROM) or a terminal control RECEIVE.) CICS has already read and translated this input, and it is too late to suppress translation. (Its arrival caused the task to be invoked, as explained in “How tasks are started” on page 142.) Consequently, ASIS is ignored entirely in pseudoconversational transaction sequences, where at most one RECEIVE MAP (without FROM) occurs per task, by definition. For the same reason, you cannot use ASIS with the FROM option (see “Formatting other input” on page 505).

Using the attention identifier

This information is part of the input in many applications, and you may also need it to interpret the input correctly.

For example, in the “quick update” transaction, we need some method for allowing the clerk to exit our transaction, and we have not yet provided for this. Suppose that we establish the convention that pressing PF12 causes you to leave control of the transaction. We would then code the following after our RECEIVE MAP command:

```
IF EIBAID = DFHPPF12,  
    EXEC CICS SEND CONTROL FREEKB ERASE END-EXEC  
    EXEC CICS RETURN END-EXEC.
```

This would end the transaction without specifying which one should be executed next, so that the operator would regain control. The SEND CONTROL command that precedes the RETURN unlocks the keyboard and clears the screen, so that the operator is ready to enter the next request.

The hexadecimal values that correspond to the various attention keys are defined in a copy book called DFHAID. To use these definitions, you simply copy DFHAID into your working storage, in the same way that you copy DFHBMSCA to use the predefined attributes byte combinations (see “Attribute value definitions: DFHBMSCA” on page 481). The contents of the DFHAID copy book are listed in the *CICS Application Programming Reference* manual.

Using the HANDLE AID command

You can also use a HANDLE AID command to identify the attention key used (unless you are writing in C or C++, which does not support HANDLE AID commands). HANDLE AID works like other HANDLE commands; you issue it before the first RECEIVE command to which it applies, and it causes a program branch on completion of subsequent RECEIVES if a key named in the HANDLE AID is used.

For example, an alternative to the “escape” code just shown would be:

```
EXEC CICS HANDLE AID PF12(ESCAPE) END-EXEC.  
...  
EXEC CICS RECEIVE MAP('QUPMAP') MAPSET('QUPSET') ...
```

```

...
ESCAPE.
EXEC CICS SEND CONTROL FREEKB ERASE END-EXEC
EXEC CICS RETURN END-EXEC.

```

HANDLE AID applies only to RECEIVE commands in the same program. The specification for a key remains in effect until another HANDLE AID in the same program supersedes it by naming a new label for the key or terminates it by naming the key with no label. A RESP, RESP2, or NOHANDLE option on a RECEIVE command exempts that particular command from the effects of HANDLE AID specifications, but they remain in effect otherwise.

If you have a HANDLE active for an AID received during an input operation, control goes to the label specified in the HANDLE AID, regardless of any exceptional condition that occurs and whether or not a HANDLE CONDITION is active for that exception. HANDLE AID can thus mask an exceptional condition if you check for it with HANDLE CONDITION. For this reason you may prefer to use an alternative test for the AID or exceptional conditions or both. You can check EIBAID for the AID and use the RESP option or check EIBRESP for exceptions. You need to be especially aware of MAPFAIL in this respect, as noted on page 504.

Finding the cursor

In some applications, you need to know where the operator left the cursor at the time of sending. There are two ways of finding out. If your map specifies CURSLOC=YES, BMS turns on the seventh (X'02') bit in the flag subfield of the map field where the cursor was left. This only works, of course, if the cursor is left in a map field to which you assigned a name.

Also, because the flag subfield is used to indicate both cursor presence and field erasure, you need to test the bits individually if you are looking for one in particular: the X'80' bit for field erasure and the X'02' bit for the cursor. If you are using a language in which it is awkward to test bits, you can test for combinations. A value of X'80' or X'82' signals erasure; either X'02' or X'82' indicates the cursor. The DFHBMSCA definitions described in the *CICS Application Programming Reference* manual include all of these combinations.

You can also determine the position of the cursor from the EIBCPOSN field in the EIB. This is the absolute position on the screen, counting from the upper left (position zero) and going across the rows. Thus a value of 41 on a screen 40 characters wide would put the cursor in the second row, second column. Avoid this method if possible, because it makes your program sensitive to the placement of fields on the screen and to the terminal type.

Processing the mapped input

To illustrate how the input subfields are used, we return to “quick update”. After we have the input, we need to do some checks on it before continuing. First, we require that the charge be entered (that is, that the input length be greater than zero), and be positive and numeric.

```

IF CHGL = 0, MOVE -1 TO CHGL
MOVE 1 TO ERR-NO
ELSE IF CHGI NOT > ZERO OR CHGI NOT NUMERIC,
MOVE DFHUNIMD TO CHGA,
MOVE -1 TO CHGL
MOVE 2 TO ERR-NO.

```

The 'MOVE -1' statements here and following put the cursor in the first field in error when we redisplay the map, as explained in “Positioning the cursor” on page 488. The message number tells us what message to put in the message area; 1 is “enter a charge”, and so on through 6, for “charge is over limit”. We do these checks in roughly ascending order of importance, to ensure that the most basic error is the one that gets the message. At the end of the checking, we know that everything is okay if ERR-NO is zero.

An account number must be entered, as well as the charge. If we have one (whatever the condition of the charge), we can retrieve the customer’s account record to ensure that the account exists:

```

IF ACCTNOL = 0, MOVE -1 TO ACCTNOL
  MOVE 3 TO ERR-NO
ELSE EXEC CICS READ FILE (ACCT) INTO (ACCTFILE-RECORD)
  RIDFLD (ACCTNOI) UPDATE RESP(READRC) END-EXEC
  IF READRC = DFHRESP(NOTFOUND), MOVE 4 TO ERR-NO,
    MOVE DFHUNMD TO ACCTNOA
    MOVE -1 TO ACCTNOL
  ELSE IF READRC NOT = DFHRESP(NORMAL) GO TO HARD-ERR-RTN.

```

If we get this far, we continue checking, until an error prevents us from going on. We need to ensure that the operator gave us a good account number (one that is not in trouble), and that the charge is not too much for the account:

```

IF ERR-NO NOT > 2
  IF ACCTFILE-WARNCODE = 'S', MOVE DFHMBRY TO MSGA
  MOVE 5 TO ERR-NO
  MOVE -1 TO ACCTNOL
  EXEC CICS LINK PROGRAM('NTFYCOPS') END-EXEC
  ELSE IF CHGI > ACCTFILE-CREDIT-LIM - ACCTFILE-UNPAID-BAL
    - ACCTFILE-CUR-CHGS
    MOVE 6 TO ERR-NO
    MOVE -1 TO ACCTNOL.
IF ERR-NO NOT = 0 GO TO REJECT-INPUT.

```

Handling input errors

As illustrated in “quick update,” above, whenever you have operator input to process, there is almost always a possibility of incorrect data, and you must provide for this contingency in your code. Usually, what you need to do when the input is wrong is:

- Notify the operator of the errors. Try to diagnose all of them at once; it is annoying to the operator if you present them one at a time.
- Save the data already entered, so that the operator does not have to rekey anything except corrections.
- Arrange to recheck the input after the operator makes corrections.

Flagging errors

In the preceding code for the “quick update” transaction, we used the message field to describe the error (the first one, anyway). We highlighted all the fields in error, provided there was any data in them to highlight, and we set the length subfields to -1 so that BMS would place the cursor in the first bad field. We send this information using the same map, as follows:

```

REJECT-INPUT.
  MOVE LOW-VALUES TO ACCTNOO CHGO.
  EXEC CICS SEND MAP('QUPMAP') MAPSET('QUPSET') FROM(QUPMAPO)
    DATAONLY END-EXEC.

```

Notice that we specify the DATAONLY option. We can do this because the constant part of the map is still on the screen, and there is no point in rewriting it there. We cleared the output fields ACCTNOO and CHGO, to avoid sending back the input we had received, and we used a different attributes combination to make the ACCTNO field bright (DFHUNIMD instead of DFHBMCRY). DFHUNIMD highlights the field and leaves the modified data tag on, so that if the operator resends without changing the field, the account number is retransmitted.

Saving the good input

The next step is to ensure that whatever good data the operator entered gets saved. One easy technique is to store the data on the screen. You do not have to do anything additional to accomplish this; once the MDT in a field is turned on, as it is the first time the operator touches the field, it remains on, no matter how many times the screen is read. Tags are not turned off until you erase the screen, turn them off explicitly with the FRSET option on your SEND, or set the attributes subfield to a value in which the tag is off.

The drawback to saving data on the screen is that all the data is lost if the operator uses the CLEAR key. If your task is conversational, you can avoid this hazard by moving the input to a safe area in the program before sending the error information and asking for corrections. In a pseudoconversational sequence, where the component tasks do not span interactions with the terminal, the equivalent is for the task that detects the error to pass the old input forward to the task that processes the corrected input. You can forward data through a COMMAREA on the RETURN command that ends a task, by writing to temporary storage, or in a number of other ways (see Chapter 13, “Sharing data across transactions,” on page 183 for possibilities).

In addition to avoiding the CLEAR key problem, storing data in your program or in a temporary storage queue reduces inbound transmission time, because you transmit only changed fields on the error correction cycles. (You must specify FRSET when you send the error information to prevent the fields already sent and not corrected from coming in again.) You can also avoid repeating field audits because, after the first time, you need to audit only if the user has changed the field or a related one.

However, these gains are at the expense of extra programming and complexity, and therefore the savings in line time or audit path length must be considerable, and the probability of errors high, to justify this choice. You must add code to merge the new input with the old, and if you have turned off the MDTs, you need to check both the length and the flag subfield to determine whether the operator has modified a map field. Fields with new data have a nonzero length; those which had data and were subsequently erased have the high-order bit in the flag subfield on.

A good compromise is to save the data both ways. If the operator clears the screen, you use the saved data to refresh it; otherwise you simply use the data coming in from the screen. You do not need any merge logic, but you protect the operator from losing time over an unintended CLEAR.

For our “quick update” code, with its minimal audits and transmissions, we choose the “do nothing” approach and save the information on the screen.

Rechecking

The last requirement is to ensure that the input data is rechecked. If your task is conversational, this simply means repeating the audit section of your code after you have received (and merged, if necessary) the corrected input. In a

pseudoconversational sequence, you usually repeat the transaction that failed. In the example, because we saved the data on the screen in such a way that corrected data is indistinguishable from new data, all we need to do is arrange to execute the same transaction against the corrected data, thus:

```
EXEC CICS RETURN TRANSID('QUPD') END-EXEC.
```

where 'QUPD' is the identifier of the “quick update” transaction.

Sending mapped output after mapped input

If your transaction makes it through its input audits and the attendant hazards, the processing specific to mapped input is complete. The next step, frequently, is to prepare and send the transaction output. In general, if the output is to be mapped, you follow the steps outlined in Chapter 33, “Sending BMS mapped output,” on page 477. However, the acquisition of storage for building the map may be affected by the input mapping you have already done. If the output and input maps are different, but in the same map set or in map sets defined to overlay one another, you have already done the storage acquisition during your input mapping process. If your output and input maps overlay one another, you need to ensure that you save any map input you still need and clear the output structure to nulls before you start building the output map. If this is awkward, you may want to define the maps so that they do not overlay one another. (See “BASE and STORAGE options” on page 478 for your choices in this regard.)

Your transaction may also call for using the same map for output as input. This is routine in code that handles input errors, as we have already seen, and also in simple transactions like “quick update”. One-screen data-entry transactions are another common example.

When you are sending new data with a map already on the screen, you can reduce transmission with the DATAONLY option, and you may need only the SEND CONTROL command. See “Merging the symbolic and physical maps” on page 484 for a discussion of these options.

For the “quick update” transaction, however, we need to fill in the message field with our “go” response (and update the file with the charge to finish our processing):

```
MOVE 'CURRENT CHARGE OKAY; ENTER NEXT' TO MSGO
ADD CHGI TO ACCTFILE-CUR-CHGS
EXEC CICS REWRITE FILE('ACCT') FROM (ACCTFILE-RECORD)....
```

We also need to erase the input fields, so that the screen is ready for the next input. We have to do this both on the screen (the ERASEAUP option erases all unprotected fields) and in the output structure (because the output subfield overlays the input subfield and the input data is still there).

```
MOVE LOW-VALUES TO ACCTNOO CHGO.
EXEC CICS SEND MAP('QUPMAP') MAPSET('QUPSET') FROM(QUPMAPO)
DATAONLY ERASEAUP END-EXEC.
```

Finally, we can return control to CICS, specifying that the same transaction is to be executed for the next input.

```
EXEC CICS RETURN TRANSID('QUPD') END-EXEC.
```

MAPFAIL and other exceptional conditions

The exceptional conditions that can occur on a RECEIVE command are all described in the *CICS Application Programming Reference* manual, and most are self-explanatory. One of them warrants discussion, however, because it can result from a simple operator error. This is MAPFAIL, which occurs when no usable data is transmitted from the terminal or when the data transmitted is unformatted (in the 3270 sense—see “Unformatted mode” on page 404). MAPFAIL occurs on a RECEIVE MAP if the operator has used the CLEAR key or one of the PA keys. It also occurs if the operator uses ENTER or a PF key from a screen where:

- No fields defined in the map have the modified data tag set on (this means the operator did not key anything and you did not send any fields with the tags already set, so that no data is returned on the read), and
- The cursor was not left in a field defined in the map and named, or the map did not specify CURSLOC=YES.

Pressing ENTER prematurely or a “short read” key accidentally is an easy mistake for the operator to make. In the interest of user friendliness, you may want to refresh the screen after MAPFAIL instead of ending the transaction in error.

MAPFAIL also occurs if you issue a RECEIVE MAP without first formatting with a SEND MAP or equivalent in the current or a previous task, and can occur if you use a map different from the one you sent. This might signal an error in logic, or it might simply mean that your transaction is in its startup phase. For instance, in our “quick update” example, we have not made any provision for getting started—that is, for getting an empty map onto the screen so that the operator can start using the transaction. We could use a separate transaction to do this, but we might as well take advantage of the code we need to refresh the screen after a MAPFAIL. What we need is:

```
IF RCV-RC = DFHRESP(MAPFAIL)
  MOVE 'PRESS PF12 TO QUIT THIS TRANSACTION' TO MSGO
  EXEC CICS SEND MAP('QUPMAP') MAPSET('QUPSET')
    FROM(QUPMAPO) END-EXEC.
```

We are reminding the operator how to escape, because attempts to do this may have caused the MAPFAIL in the first place. If we had not wanted to send this message, or if it was the default in the map, we could have used the MAPONLY option:

```
EXEC CICS SEND MAP('QUPMAP') MAPSET('QUPSET') MAPONLY END-EXEC.
```

When MAPFAIL occurs, the input map structure is not cleared to nulls, as it is otherwise, so it is important to test for this condition if your program logic depends on this clearing.

You can issue a HANDLE CONDITION command to intercept MAPFAIL, as you can other exception conditions. If you do, and you also have a HANDLE AID active for the AID you receive, however, control goes to the label specified for the AID and not that for MAPFAIL, as explained in “Using the HANDLE AID command” on page 499. In this situation you will be unaware of the MAPFAIL, even though you issued a HANDLE for it, unless you also test EIBRESP.

EOC condition

EOC is another condition that you encounter frequently using BMS. It occurs when the end-of-chain (EOC) indicator is set in the request/response unit returned from VTAM. EOC does not indicate an error, and the BMS default action is to ignore this condition.

Formatting other input

Although the data that you format with a RECEIVE MAP command normally comes from a terminal, you can also format data that did not come from a terminal, or that came indirectly. For example, you might not know which map to use until you receive the input and inspect some part of it. This can happen when you use special hardware features like partitioning or logical device codes, and also in certain logic situations. You might also need to format data that was read from a formatted screen by an intermediate process (without mapping) and later passed to your transaction.

The FROM option of the RECEIVE MAP command addresses these situations. FROM tells BMS that the data has already been read, and only the translation from the native input stream to the input map structure is required.

Because the input has already been read, you need to specify its length if you use FROM, because BMS cannot get this information from the access method, as it does normally. If the data came originally from a RECEIVE command in another task, the length on the RECEIVE MAP FROM command should be the length produced by that original RECEIVE.

For the same reason, you cannot suppress translation to upper case with the ASIS option when you use FROM. Moreover, BMS does not set EIBRID and EIBCURSR after a RECEIVE FROM command.

And finally, BMS does not know from what device the input came, and it assumes that it was your current principal facility. (You cannot even use RECEIVE FROM without a principal facility, even though no input/output occurs.) If the data came from a different type of device, you have to do the mapping in a transaction with a similar principal facility to get the proper conversion of the input data stream.

Note: You cannot map data read with a terminal control RECEIVE with the BUFFER option, because the input data is unformatted (in the 3270 sense). If you attempt to RECEIVE MAP FROM such input, MAPFAIL occurs.

Chapter 36. BMS logical messages

The disposition options do not affect the correspondence between SEND MAP commands and pages of output. You get one page for each SEND MAP command, unless you also use a second feature of full BMS, the ACCUM option. ACCUM allows you to build pages piecemeal, using more than one map, and like PAGING, it allows your message to exceed a page. You do not have to worry about page breaks or about tailoring your output to a specific page or screen capacity. BMS handles these automatically, giving you control at page breaks if you wish. Details on cumulative page building are in Chapter 37, “Cumulative output — the ACCUM option,” on page 513.

As soon as you create an output message of more than one page, or a single page composed of several different maps, you are doing something BMS calls **cumulative** mapping. PAGING implies multiple pages, and ACCUM implies both multiple and composite pages, and so at the first appearance of either of these options, BMS goes into cumulative mapping mode and begins a **logical message**. The one-to-one correspondence between SEND commands and messages ends, and subsequent SEND MAPS simply add to the current logical message. Individual pages within the message are still disposed of as soon as they are complete, but they all belong to the same logical message, which continues until you tell BMS to end it.

This chapter describes:

- “Building logical messages”
- “The SEND PAGE command” on page 508
- “RETAIN and RELEASE” on page 508
- “The AUTOPAGE option” on page 510
- “Terminal operator paging: the CSPG transaction” on page 510
- “Logical message recovery” on page 511

Building logical messages

When you start a logical message, you need to observe a number of rules:

- You can build only one logical message at a time. If you are routing this message, BMS may create more than one logical message internally, but in terms of content, there is only one. After you complete the message and dispose of it, you can build another in the same task, using different options if you wish.
- Options related to message management must be the same on all commands that build the message. These are:
 - the disposition option: PAGING, TERMINAL, or SET
 - the option governing page formation: ACCUM should be present on all commands or absent on all
 - the identifier for the message in CICS temporary storage: the REQID option value.

Switching options mid-message results in the INVREQ condition or, in the case of REQID, the IGREQID condition.

- The ERASE, ERASEAUP, NLEOM, and FORMFEED options are honored if they are used on *any* of the BMS commands that contribute to the page.
- The values of the CURSOR, ACTPARTN, and MSR options for the page are taken from the most recent SEND MAP command, if they are specified there, and from the map if not.

- The 3270 write control character (WCC) from the most recent SEND MAP command is used. The WCC is assembled from the ALARM, FREEKB, PRINT, FRSET, L40, L64, L80, and HONEOM options in the command whenever *any* of them is specified. Otherwise, it is built from the same options in the map; options from the command are never mixed with those in the map.
- The FMHPARMS from all commands used to build the message are included.
- You can use both SEND MAP and SEND CONTROL commands to build a logical message, as long as the options noted above are consistent. You can also build a logical message with a combination of SEND TEXT and SEND CONTROL commands. (SEND TEXT is an alternative to SEND MAP for formatting text output, covered in “The SEND TEXT command” on page 521.) However, you cannot mix SEND MAP and SEND TEXT in the same message unless you are using partitions or logical device codes, subjects covered in Chapter 41, “Partition support,” on page 541 and “Logical device components” on page 549 respectively.

There are also two special forms of SEND TEXT which allow combined mapping and text output, but to which other restrictions apply. See “SEND TEXT MAPPED and SEND TEXT NOEDIT” on page 525 for details.

- While you are building a logical message, you can still converse with your terminal. You cannot use BMS commands to write to the terminal unless you are also routing, but you can use BMS RECEIVE MAP commands and terminal control SEND and RECEIVE commands.

The SEND PAGE command

When you have completed a logical message, you notify BMS with a SEND PAGE command. If you used the ACCUM option, SEND PAGE causes BMS to complete the current page and dispose of it according to the disposition option you established, as it has already done for any previous pages. If your disposition is TERMINAL, this last page is written to your principal facility; if SET, it is returned to the program; and if PAGING, it is written to temporary storage. If your disposition was PAGING, BMS also arranges delivery of the entire message to your principal facility. Options on the SEND PAGE command govern how this is done, as explained in RETAIN and RELEASE.

A SYNCPOINT command or the end of your task also ends a logical message, implicitly rather than explicitly. Where possible, BMS behaves as if you had issued SEND PAGE before your SYNCPOINT or RETURN, but you lose the last page of your output if you used the ACCUM option. Consequently, you should code SEND PAGE explicitly.

You also can delete an incomplete logical message if for some reason you decide not to send it. You use the PURGE MESSAGE command in place of SEND PAGE. PURGE MESSAGE causes BMS to delete the current logical message and associated control blocks, including any pages already written to CICS temporary storage. You can create other logical messages subsequently in the same task, if you wish.

RETAIN and RELEASE

When you complete a logical message with a disposition of PAGING, BMS arranges to deliver the entire logical message, which it has accumulated in temporary storage. The display or printing of pages can be done inline, immediately after the SEND PAGE command, but it is more common to schedule a separate task for the purpose. In either case, CICS supplies the programs required. These

programs allow a terminal operator to control the display of the message, paging back and forth, displaying particular pages, and so on. When a separate task is used, it executes pseudoconversationally under transaction code CSPG. When the display is inline, the work is done (by the same CICS-supplied programs) within the task that created the message, which becomes conversational as a result.

You indicate how and when the message is sent by specifying RETAIN, RELEASE, or neither on your SEND PAGE command. The most common choice, and the default, is neither. It causes CICS to schedule the CICS-supplied transaction CSPG to display the message and then return control to the task. The CSPG transaction is queued with any others waiting to be executed at your terminal, which execute in priority sequence as the terminal becomes free. In the ordinary case, where no other tasks waiting, CSPG executes as soon as the creating task ends.

Note: The terminal must be defined as allowing automatic transaction initiation for CICS to start CSPG automatically (ATI(YES) in the associated TYPETERM definition). If it is not, the operator has to enter the transaction code CSPG or one of the paging commands to get the process started when neither RELEASE nor RETAIN is specified.

The RELEASE option works similarly, but your task does not regain control after SEND PAGE RELEASE. Instead, BMS sends the first page of the message to the terminal immediately. It then ends your task, as if a CICS RETURN had been executed in the highest level program, and starts a CSPG transaction at your terminal so that the operator can display the rest of the pages. The CSPG code executes pseudoconversationally, just as it does if you specify neither RELEASE nor RETAIN, and the original task remains pseudoconversational if it was previously.

There are two other distinctions between RELEASE and using neither option:

- RELEASE allows you to specify the transaction identifier for the next input from the terminal, after the operator is through displaying the message with CSPG.
- RELEASE also permits the terminal operator to chain the output from multiple transactions (see “Terminal operator paging: the CSPG transaction” on page 510).

SEND PAGE RETAIN causes BMS to send the message immediately. When this process is complete, your task resumes control immediately after the SEND PAGE command. When the terminal is a display, BMS provides the same operator facilities for paging through the message as the CSPG transaction does, but within the framework of your task. The code that BMS uses for this purpose issues RECEIVE commands to get the operator’s display requests, and this causes your task to become conversational.

Note: If an error occurs during the processing of a SEND PAGE command, the logical message is not considered complete and no attempt is made to display it. BMS discards the message in its cleanup processing, unless you arrange to regain control after an error. If you do, you can either delete the logical message with a PURGE command or retry the SEND PAGE. You should not retry unless the error that caused the failure has been remedied.

The AUTOPAGE option

Your SEND PAGE command also tells BMS how to deliver the pages to the terminal. For display terminals, you want CSPG to send one page at a time, at the request of the terminal operator. For printers, you want to send one page after another. You control this with the AUTOPAGE or NOAUTOPAGE option on your SEND PAGE command. NOAUTOPAGE lets the terminal operator control the display of pages; AUTOPAGE sends the pages in ascending sequence, as quickly as the device can accept them. If you specify neither, BMS determine which is appropriate from the terminal definition.

Note: If your principal facility is a printer, you can sometimes use a disposition of TERMINAL rather than PAGING, because successive sends to a printer do not overlay one another as they do on a display. TERMINAL has less overhead, especially if you do not need ACCUM either, and thus avoid creating a logical message.

Terminal operator paging: the CSPG transaction

The CICS-supplied paging transaction, CSPG, allows a user at a terminal to display individual pages of a logical message by entering page retrieval requests. Your systems staff define the transaction identifiers for retrieval and other requests supported by CSPG in the system initialization table; sometimes program function keys are used to minimize operator effort.

Retrieval can be sequential (next page or previous page) or random (a particular page, first page, last page). In addition to page retrieval, CSPG supports the following requests:

Page copy

Copy the page currently on display to another terminal. BMS reformats the page if the target terminal has a different page size or different formatting characteristics, provided the terminal is of a type supported by BMS.

Message query

List the messages waiting to be displayed at the terminal with CSPG. The list contains the BMS-assigned message identifier and, for a routed message, the message title, if the sender provided one.

Purge message

Delete the logical message.

Page chaining

Suspend the current CSPG transaction after starting to display a message, execute one or more other transactions, and then resume the original CSPG display. An intervening transaction may itself produce BMS or terminal output. If this output is a BMS logical message created with the PAGING and RELEASE or RETAIN options, this message is “chained” to the original one, and the operator can switch between one and the other.

Switch to autopage

Switch from NOAUTOPAGE display mode to AUTOPAGE mode. For terminals that combine a keyboard and hard copy output, this allows an operator to purge or print a message based on inspection of specific pages.

The process of examining pages continues until the operator signals that the message can be purged. CSPG provides a specific request for this purpose, as

noted above. If the SEND PAGE command contained the option OPERPURGE, this request is the only way to delete the message and get control back from CSPG.

If OPERPURGE is not present, however, any input from the terminal that is not a CSPG request is interpreted as a request to delete the message and end CSPG. If the message was displayed with the RETAIN option, the non-CSPG input that terminates the display can be accessed with a BMS or terminal control RECEIVE when the task resumes execution. See the *CICS Supplied Transactions* manual for detailed information about the CSPG transaction.

Logical message recovery

Logical messages created with a disposition of PAGING are kept in CICS temporary storage between creation and delivery. BMS constructs the temporary storage queue name for a message from the 2-character REQID on the SEND commands, followed by a six-position number to maintain uniqueness. If you do not specify REQID, BMS uses a value of two asterisks (**).

Temporary storage can be a recoverable resource, and therefore logical messages with a disposition of PAGING can be recovered after a CICS abend. In fact, because CICS bases the recoverability of temporary storage on generic queue names, you can make some of your messages recoverable and others not, by your choice of REQID for the message. The conditions under which logical messages are recoverable are described in the *CICS Recovery and Restart Guide*.

Routed messages are eligible for recovery, as well as messages created for your principal facility. We explain routing in Chapter 39, “Message routing,” on page 527.

Chapter 37. Cumulative output — the ACCUM option

The ACCUM option allows you to build your output cumulatively, from any number of SEND MAP commands and less-than-page-size maps. Without it, each SEND MAP command corresponds to one page (if the disposition is PAGING), or a whole message (if TERMINAL or SET). With ACCUM, however, BMS formats your output but does not dispose of it until either it has accumulated more than fits on a page or you end the logical message. You can intercept page breaks if you wish, or you can let BMS handle them automatically.

Page size is determined by the PAGESIZE or ALTPAGE value in the terminal definition. PAGESIZE is used if the PROFILE under which your transaction is running specifies the default screen size, and ALTPAGE is used if it indicates alternate screen size. (Unlike screen size, page size is not affected by the DEFAULT and ALTERNATE options that you can include with the ERASE command.)

This chapter describes:

- “Floating maps: how BMS places maps using ACCUM”
- “Page breaks: BMS overflow processing” on page 514
- “Map placement rules” on page 515
- “Input from a composite screen” on page 517
- “Performance considerations” on page 517

Floating maps: how BMS places maps using ACCUM

In our example map on page 461, we described placing maps on a screen or page absolutely, by specifying the number of the line and column for the upper left corner. However, maps can **float**. That is, they can be positioned relative to maps already written to the same page and to any edge of the page. Floating maps save program logic when you need to support multiple screen sizes or build pages piecemeal out of headers, detail lines and trailers, where the number of detail lines depends on the data.

The BMS options that allow you to do this are:

- JUSTIFY
- HEADER and TRAILER
- Relative values (NEXT and SAME) for the LINE and COLUMN options

When you are building a composite screen with the ACCUM option, the position on the screen of any particular map is determined by:

- The space remaining on the screen at the time it is sent
- The JUSTIFY, LINE and COLUMN option values in the map definition

The space remaining on the page, in turn, depends on:

- Maps already placed on the current page.
- Whether you are participating in “overflow processing”, that is, the processing that occurs at page breaks. If you are, the sizes of the trailer maps in your map sets are also a factor.

The placement rules we are about to list apply even if you do not specify ACCUM, although JUSTIFY values of FIRST and LAST are ignored. However, without ACCUM, each SEND MAP corresponds to a separate page, and thus the space remaining on the page is always the whole page.

Page breaks: BMS overflow processing

When you build a mapped logical message, you can ask BMS to notify you when a page break is about to occur; that is, when the map you just sent does not fit on the current page. This is very useful when you are forming composite pages with ACCUM. It allows you to put trailer maps at the bottom of the current page and header maps at the top of the next one, number your pages, and so on.

BMS gives your program control at page breaks if either:

- You have issued a `HANDLE CONDITION` command naming a label for the `OVERFLOW` condition, or
- You specify the `NOFLUSH` option with either the `RESP` or the `NOHANDLE` option on your `SEND MAP` commands.

Either of these actions has two effects:

- The calculation of the space remaining on the page changes. Unless the map you are sending is itself a trailer map, BMS assumes that you eventually want one on the current page. It therefore reserves space for the largest trailer in the same map set. (The largest trailer map is the one containing the `TRAILER` option that has the most lines.) If you do not intercept page breaks (or if you send a trailer map), BMS uses the true end of the page to determine whether the current map fits.
- The flow of control changes if the map does not fit on the current page. On detecting this situation, BMS raises the `OVERFLOW` condition. Then it returns control to your task *without* processing the `SEND MAP` command that caused the overflow. Control goes to the location named in the `HANDLE CONDITION` command if you used one. With `NOFLUSH`, control goes to the statement after the `SEND MAP` as usual, and you need to test the `RESP` value or `EIBRESP` in the `EIB` to determine whether overflow occurred.

When your program gets control after overflow, it should:

- Add any trailer maps that you want on the current page. BMS leaves room for the one with the most lines in the map set you just used. If this is not the right number of lines to reserve, or if you are using several map sets, you can ensure the proper amount by including a dummy map in any map set that may apply. The dummy map must specify `TRAILER` and contain the number of lines you wish to reserve; you do not need to use it in any `SEND MAP` commands.
- Write any header maps that you want at the top of the next page.
- Resend the map that caused the overflow in the first place. You must keep track of the data and map name at the time the overflow occurs; BMS does not save this information for you. Note that if you have several `SEND MAP` commands which might cause overflow, the program logic required to determine which one you need to reissue is more complex if you use `HANDLE CONDITION OVERFLOW` than if you use `NOFLUSH`.

Once `OVERFLOW` is turned on, BMS suspends returning control to your program when the output does not fit on the current page, although it still uses overflow rules for calculating the remaining space. `OVERFLOW` remains on until BMS processes the first `SEND MAP` naming a map which is *not* a header or a trailer. This allows you to send your trailers and headers without disabling your `HANDLE CONDITION OVERFLOW` or changing your response code tests, and reinstates your overflow logic as soon as you return to regular output. (Resending the map that originally caused overflow is usually the event that turns off the overflow condition.)

If you do not intercept overflows, BMS does not notify your program when a page break occurs. Instead, it disposes of the current page according to the disposition option you have established and starts a new page for the map that caused the overflow.

Map placement rules

The primary placement of maps on the screen is from top to bottom. You can place maps side-by-side where space permits, provided you maintain the overall flow from top to bottom. The precise rules for a given SEND MAP ACCUM command are as follows:

1. The highest line on which the map might start is determined as follows:
 - a. If the map definition contains JUSTIFY=FIRST, BMS goes immediately to a new page (at Step 5), unless the only maps already on the page are headers placed there during overflow processing. In this case, BMS continues at Step 1c.
 - b. If the map specifies JUSTIFY=LAST, BMS starts the map on the lowest line that allows it to fit on the page. If the map is a trailer map or you are not intercepting overflows or you are already in overflow processing, BMS uses all the space on the page. Otherwise, BMS places the map as low on the page as it can while still retaining room for the largest trailer map. If the map fits vertically using this starting line, processing continues at Step 3 (the LINE option is ignored if JUSTIFY=LAST); if not, overflow occurs (Step 5).

Note: JUSTIFY=BOTTOM is the same as JUSTIFY=LAST for output operations with ACCUM. (There are differences without ACCUM and for input mapping; see the *CICS Application Programming Reference manual*).

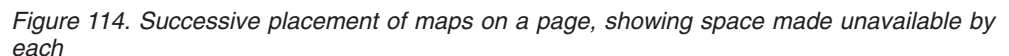
- c. If there is no vertical JUSTIFY value (or after any overflow processing caused by JUSTIFY=FIRST has been completed), the LINE operand is checked. If an absolute value for LINE is given, that line is used, provided it is at or below the starting line of the map most recently placed on the page. If the value is above that point, BMS goes to a new page at Step 5.

If LINE=NEXT, the first completely unused line (below all maps currently on the page) is used. If LINE=SAME, the starting line of the map sent most recently is used.
2. BMS now checks that the map fits vertically on the screen, given its tentative starting line. Here again, BMS uses all of the space remaining if the map is a trailer map, if you are not intercepting overflows or if you are already in overflow processing. Otherwise, BMS requires that the map fit and still leave space for the largest trailer map. If the map does not fit vertically, BMS starts a new page (Step 5).
3. Next, BMS checks whether the map fits horizontally, assuming the proposed starting line. In horizontal positioning, the JUSTIFY option values of LEFT and RIGHT come into play. LEFT is the default, and means that the COLUMN value refers to the left-hand side of the map. A numeric value for COLUMN tells where the left edge of the map should start, counting from the left side of the page. COLUMN=NEXT starts the map in the first unused column from the left on the starting line. COLUMN=SAME means the left-hand column of the map most recently placed on the screen which also specified JUSTIFY=LEFT and which was not a header or trailer map.

JUSTIFY=RIGHT means that the COLUMN value refers to the right-hand edge of the map. A numeric value tells where the right edge of the map should start, *counting from the right*. COLUMN=NEXT means the first available column from

If the map does not fit horizontally, BMS adjusts the starting line downward, one line at a time, until it reaches a line where the map does fit or overflow occurs. Processing resumes with the vertical check (Step 2) after each adjustment of the starting line.

- Figure 114 shows how the remaining space is reduced with each new map placed.



- When the current map does not fit on a page, BMS determines whether it should return control to your program. If you have asked for control at overflow and you are not already in overflow processing, BMS returns control as described in “Page breaks: BMS overflow processing” on page 514. Otherwise, BMS disposes of the current page according to the disposition option you have established, starts a new page, and resumes processing for the map that would not fit at Step 1.

ASSIGN options for cumulative processing

To help you manage the complexities of building a composite screen, CICS provides ASSIGN command options that relate specifically to cumulative processing:

MAPCOLUMN
MAPHEIGHT
MAPLINE
MAPWIDTH

All apply to the map most recently sent. MAPHEIGHT and MAPWIDTH are the size (number of rows and columns) and MAPLINE and MAPCOLUMN are the origin of the map (the position of the upper left corner).

Input from a composite screen

You can read mapped input from a screen built from multiple maps, but there are restrictions. First, you can specify only one map in your RECEIVE MAP command, whereas the screen may have been written with several.

Second, BMS cannot know how to position a floating map for input and assumes that the map in your RECEIVE MAP command was written to an empty screen. Thus LINE or COLUMN values of NEXT or SAME are interpreted differently on input than on output. JUSTIFY=LAST is ignored altogether; you should use JUSTIFY=BOTTOM if you want to place a map at the bottom of the screen and read data back from it. See the *CICS Application Programming Reference* manual for the exact rules.

Performance considerations

There are three components to the overall efficiency of the part of your application that the end user sees: processor path length, communications line utilization, and user time. Path length and line time used to be paramount, and much design and programming effort has been invested in minimizing them.

As online systems have evolved, however, the emphasis has shifted steadily to making things as easy, pleasant and quick for the user as possible, at the expense of the other factors if necessary. Also, as processors have become cheaper, designers have been willing to expend cycles to reduce programming and maintenance effort as well as to make the user interface better.

We have already given you references on elements of good design for the user interface, in “Personal computers” on page 388, and usually these should be your overriding considerations. In this section, we point out some ways that you can reduce path length and line time as well. You need to judge for yourself whether and to what extent the savings justify extra programming effort or a less general design.

Minimizing path length

Ordinarily, the number of instructions executed in a single CICS command is large in comparison to the number of instructions in the application program that invoked it. Consequently, the path length for a given task ordinarily depends more on the number and type of CICS commands than on anything else, and commands are the most fertile area for tuning. Commands vary by type, of course, and path length for any given command may vary considerably with circumstances.

For BMS, some recommendations are:

- Build screens (pages) with a single command when practical. Avoid building a composite screen with the ACCUM feature when a modest amount of additional programming accomplishes the same function, and avoid building a composite screen by multiple physical writes, as described in “Outside the map” on page 487, except in unusual circumstances.
- Avoid producing more output at one time than the user is likely to inspect. Some transactions—inquiries, especially—produce many pages of output for certain input values. When this happens, the user usually narrows the search and repeats the inquiry, rather than page through the initial output. To avoid the path length of producing output that is never viewed, you can limit it to some reasonable number of pages, inform the user on the last page that there is more, and save the information required to restart the search from that point if the user requests it. The extra programming is minimal; see Chapter 13, “Sharing data across transactions,” on page 183 for ways to save the restart data.
- Use commands that are on the BMS “fast path” if possible. (See “Minimum BMS” on page 453 for the commands and terminal types that qualify.)
- Use terminal control commands for very simple inputs and outputs, where you do not need BMS formatting or other function. If path length is critical, you may want to use terminal control entirely. However, the advantages of BMS over terminal control in terms of flexibility, initial programming effort and maintainability are significant, and usually outweigh the path length penalty.

Reducing message lengths

You can take advantage of 3270 hardware to reduce the length of both inbound and outbound messages. If the bandwidth in any link between the terminal and the processor is constrained, you get better response overall with shorter messages. However, the time for any given transmission depends on the behavior of other users of those links at the time, and so you may not see improvement directly. Here are some of the possibilities for reducing the length of a 3270 datastream:

- Avoid turning on MDTs unnecessarily when you send a screen, because they cause the associated input fields to be transmitted on input. Ordinarily, you do not need to set the tag on, because the hardware does this when the user enters input into the field. The tag remains on, no matter how many times the screen is transmitted, until explicitly turned off by program (by FRSET, ERASEAUP, or ERASE, or by an override attribute byte with the tag off). The only time you need to set it on by program is when you want to store data on the screen in a field that the user does not modify, or when you highlight a field in error and you want the field returned whether or not the user changes it. In this case you need to turn on the MDT as well as the highlighting.
- Use FRSET to reset the MDTs when you do not want input on a screen retransmitted (that is, when you have saved it and the user does not need to change it on a subsequent transmission of the same screen). (See “Saving the good input” on page 502 for more.)
- Do not initialize input fields to blanks when you send the screen because, on input, blanks are transmitted and nulls are not. Hence the data stream is shortened by the unused positions in each modified field if you initialize with nulls. The appearance on the screen is the same, and the data returned to the program is also the same, if you map the input.
- For single-screen data entry operations, use ERASEAUP to clear data from the screen, rather than resending the screen.
- If you are updating a screen, send only the changed fields, especially if the changes are modest, as when you highlight fields in error or add a message to

the screen. In BMS, you can use the DATAONLY option, both to shorten the data stream and reduce the path length (see “DATAONLY option” on page 485). To highlight a field, in fact, you send only the new attribute byte; the field data remains undisturbed on the screen.

- If you are using terminal control commands, format with set buffer address (SBA) and repeat-to-address (RA) orders, rather than spacing with blanks and nulls. (BMS does this for you.)

Chapter 38. Text output

If the output you are sending to the terminal is simply text, and you do not need to format the screen for subsequent input, you do not need to create a map. BMS provides a different command expressly for this purpose: SEND TEXT, which formats without maps.

When you use SEND TEXT, BMS breaks your output into pages of the proper width and depth for the terminal to which it is directed. Lines are broken at word boundaries, and you can add header and trailer text to each page if you wish. Page size is determined as it is for other BMS output (see “The SEND PAGE command” on page 508).

This chapter describes:

- “The SEND TEXT command”
- “Text pages” on page 522
- “Text lines” on page 523
- “Header and trailer format” on page 524
- “SEND TEXT MAPPED and SEND TEXT NOEDIT” on page 525

The SEND TEXT command

Except for the different type of formatting performed, the SEND TEXT command is very similar to SEND MAP. You specify the location of the text to be formatted in the FROM option and its length in the LENGTH option. Nearly all the options that apply to mapped output apply to text output as well, including:

Device controls

FORMFEED, ERASE, PRINT, FREEKB, ALARM, CURSOR.

Formatting options

NLEOM, L40, L64, L80, HONEOM.

Disposition options

TERMINAL, PAGING, SET.

Page formation option

ACCUM.

In general, these options have the same meaning on a SEND TEXT command as they do on a SEND MAP command. The SEND TEXT command itself requires **standard** BMS; options like ACCUM, PAGING and SET that require **full** BMS in a mapped environment also require full BMS in a text environment.

There are also options for SEND TEXT that correspond to functions associated with the map in a SEND MAP context. These are HEADER, TRAILER, JUSTIFY, JUSFIRST and JUSLAST. We explain how they work in “Text pages” on page 522.

Two SEND MAP options that do not carry over to SEND TEXT are ERASEAUP and NOFLUSH. ERASEAUP does not apply because text uses fields only minimally, and NOFLUSH does not apply because BMS does not raise the OVERFLOW condition on text output.

Text logical messages

The presence of either the ACCUM or PAGING option on a SEND TEXT command signals BMS that you are building a logical message, just as it does in a SEND

MAP command. Text logical messages are subject to the same rules as mapped logical messages (see page 507). In particular, you can use both SEND TEXT and SEND CONTROL commands to build your message, but you cannot mix in SEND MAPs, except as noted there. You also end your message in the same way as a mapped message (see Chapter 36, “BMS logical messages,” on page 507).

Text pages

Page formation with SEND TEXT is somewhat different from page formation with SEND MAP. First, a single SEND TEXT command can produce more output than fits on a screen or a printer page (SEND MAP never does this). BMS sends the whole message, which means that you can deliver a multi-page message to a printer without using logical facilities. You cannot use the same technique for displays, however, because even though BMS delivers the whole message, the component screens overlay one another, generally too quickly for anyone to read.

If you specify ACCUM, BMS breaks the output into pages for you, and the second difference is that unless you specify a disposition of SET, your task does not get control at page breaks. Instead, when the current page has no more room, BMS simply starts a new one. It adds your header and trailer, if any, automatically, and does not raise the OVERFLOW condition. This is true whether you produced the pages with a single SEND TEXT command or you built the message piecemeal, with several. The only situation in which your task is informed of a page break is when the disposition is SET. In this case, BMS raises the RETPAGE condition to tell you that one or more pages are complete, as explained in “Using SET” on page 490.

Here are the details of how BMS builds pages of text with ACCUM:

1. Every message starts on page 1, which is initially empty.
2. If you specify the HEADER option, BMS starts every page with your header text. BMS numbers your pages in the header or trailer if you wish. (Header format and page numbering are explained on page 524.)
3. If you specify one of the justification options (JUSTIFY, JUSFIRST, JUSLAST), BMS starts your text on the indicated line. JUSFIRST begins your text on the first line after the header, or the top line if there is no header. JUSTIFY=*n* starts your text on line *n*, and JUSLAST starts it on the lowest line that allows both it and your trailer (if any) to fit on the current page. If the contents of the current page prevent BMS from honoring the justification option there, BMS goes to a new page first, at step 6.

Justification applies only to the start of the data for each SEND TEXT command; when the length of your data requires an additional page, BMS continues your text on it in the first available position there.

4. If you do not specify justification, BMS starts your text in the first position available. On the first SEND TEXT of the message, this works out the same as JUSFIRST. Thereafter, your text follows one character after the text from the previous SEND TEXT of the current logical message. (The intervening character is an attributes byte on 3270 terminals, a blank on others.)
5. Having determined the starting position, BMS continues down the page, breaking your data into lines as explained in “Text lines” on page 523, until it runs out of space or data. If you have specified a trailer, the available space is reduced by the requirement for the trailer. If the data is exhausted before the space, processing ends at this point. The message is completed when you indicate that you are finished with a SEND PAGE or PURGE MESSAGE command.

6. If your text does not fit on the current page, BMS completes it by adding your trailer text, if any, at the bottom and disposes of it according to the disposition option you have established (TERMINAL, PAGING, or SET), just as it does for a mapped logical message. The trailer is optional, like the header; you use the TRAILER option to specify it (see “Header and trailer format” on page 524).
7. BMS then goes to a new page and repeats from step 2 with the remaining data.

Text lines

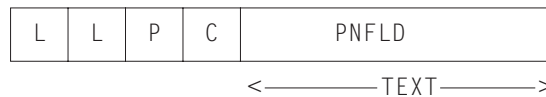
In breaking the text into lines, BMS uses the following rules:

1. Ordinarily, each line starts with what appears to be a blank. On a 3270 device, this is the attributes byte of a field that occupies the rest of the line on the screen or printed page. For other devices, it is simply a blank or a carriage control character.

An exception occurs if the task creating the output is running under a PROFILE that specifies PRINTERCOMP(YES) and the output device is a 3270 printer. In this case, no character is reserved at the beginning of each line. See “PRINTERCOMP option” on page 412.
2. BMS copies your text character for character, including all blanks, with two exceptions that occur at line end:
 - If a line ends in the middle of a word, BMS fills out the current line with blanks and places the word that would not fit in the first available position of the next line. For this purpose, a “word” is any string of consecutive nonblank characters.
 - If two words are separated by a single blank, and first one fits on the current line without leaving room for the blank, the blank is removed and the next line starts at the beginning of the second word.
3. You can embed new-line (NL) characters and other print format orders as well as blanks to control the format, if the destination terminal is a printer. NLs and tabs are particularly useful with columnar data, and BMS does not filter or even interpret these characters. However, print format orders do not format displays; see “CICS 3270 printers” on page 408 for more information about using them.
4. You can also include set attribute (SA) order sequences in your output. (Each one sets the attributes of a single character in the data stream, as explained in “The set attribute order” on page 398.) BMS abends the task unless SA sequences are exactly three bytes long and represent a valid attribute type. However, if you use a valid SA sequence to a terminal that does not support the attribute, BMS removes the SA sequence and then sends the message. Attributes set with SA orders remain until overridden by subsequent orders or until another SEND TEXT command, which resets them to their default values. You should not include 3270 orders other than SA in your text. BMS treats them as display data and they do not format as intended; they may even cause a terminal error.

Header and trailer format

To place a header on the pages of a text message, you point to a block of data in the following format in the **HEADER** option:
You use the same format for trailer text, but you point to it with the **TRAILER** option.



Here:

- LL** is the length of the header (trailer) data, not including the four bytes of LL, P, and C characters. LL should be expressed in halfword binary form.
- P** is the page-number substitution character (see PNFLD below). Use a blank if you do not want page numbers.
- C** is a reserved 1-byte field.
- TEXT** is the header (trailer) text to be placed at the top (bottom) of each page of output. Use new-line characters (X'15') to indicate where line breaks should occur if you want multiple lines.

PNFLD

is the page number field within your header (trailer) text. If you want to number the pages of your output, choose a character that does not otherwise appear in your header (trailer) text. Place this character in the positions where the page number is to appear. You can use from one to five adjacent positions, depending on how large you expect your page numbers to get (32,767 is the maximum BMS allows). Place the same character in the P field above, to tell BMS where to make the substitution. Do not use X'0C', X'15', X'17', X'26', or X'FF' for P; these values are reserved for other purposes. If you do not want page numbering, simply place a blank (X'40') in P.

When you are building a logical message, you should repeat your **HEADER** and **TRAILER** options on each **SEND TEXT** command, so that they are present when the page breaks occur, and you need to specify the trailer again on the **SEND PAGE** command that terminates the message.

Here is an example of a COBOL definition for a header that simply numbers the pages, leaving room for a number up to 99.

```
EXEC CICS SEND TEXT FROM (OUTPUT-AREA)
      HEADER(HEADER-TEXT) PAGING ACCUM END-EXEC.
```

where:

```
01 HEADER-TEXT
   02 HEADER-LL          PIC S9(4) COMP VALUE +11.
   02 HEADP              PIC X VALUE '@'.
   02 FILLER              PIC X VALUE LOW-VALUE.
   02 HEADING            PIC X(11) VALUE 'PAGE NO. @@'.
```

Screens built with **SEND TEXT** are not designed for extensive input from the terminal operator. However, you can interpret the attention identifier and read simple inputs—such as those used in the CSPG transaction to control the page display—if the field structure on the screen is suitable and the operator knows or can see what is expected. (A new field starts at each line, as well as at the first character of the

text sent with each SEND TEXT command that made up the message. The fields defined are unprotected, alphameric and normal intensity, so that the operator can key into them.) Normally a terminal control RECEIVE is used in this situation; you can use RECEIVE MAP only if you can build a map with a field structure matching that of the screen.

SEND TEXT MAPPED and SEND TEXT NOEDIT

BMS provides two special forms of the SEND TEXT command that allow you to use some of the message delivery facilities of BMS for output that is already formatted. SEND TEXT MAPPED sends a *page* of device-dependent data *previously built by BMS* and captured with the SET option. You may have used either SEND MAP or SEND TEXT commands to build the page originally. See “Using SET” on page 490 for details.

SEND TEXT NOEDIT is similar, but is used to send a page of device-dependent output built by the program or some method other than BMS.

You can deliver such pages to your own principal facility individually, using a disposition of TERMINAL, or you can include them in a logical message built with the PAGING option. In a logical message, these forms can be mixed with ordinary SEND TEXT commands or with SEND MAP⁷ commands, as long as each BMS SEND represents a separate page (that is, the ACCUM option is not used).

You can also use these commands in a routing environment (described in Chapter 39, “Message routing,” on page 527). Whether you are routing or sending to your own terminal, you must ensure that the data stream is appropriate to the destinations; BMS does not check before transmission, other than to remove 3270 attributes that the destination does not support.

None of the page-formatting options, ACCUM, JUSTIFY, JUSFIRST, JUSLAST, HEADER, and TRAILER, apply to either of these commands, because the page is already formatted and built, by definition.

The primary difference between the MAPPED and NOEDIT forms is that SEND TEXT MAPPED uses the 4-byte page control area (PGA) that BMS appends to the end of pages returned by the SET option. This area tells BMS the write command and write control character to be used, which extended attributes were used on the page, and whether the page contains formfeeds, data in SCS format, 14- or 16-bit buffer addresses, structured fields and FMHs. It allows BMS to deliver a page built for one device to one with different hardware characteristics, as might be required for a page copy or a routing operation. With SEND TEXT NOEDIT, you specify this type of information on the command itself. You should use SEND TEXT MAPPED for output created with BMS, and NOEDIT for output formatted by other means. You cannot include structured fields in the output with either SEND TEXT MAPPED or SEND TEXT NOEDIT, incidentally; you must use a terminal control SEND for such output.

The LENGTH option for a SEND TEXT MAPPED command should be set from the TIOATDL value returned when the page was built, which does not include the PGA (see “Using SET” on page 490). If you copy the page for later use with SEND TEXT MAPPED, however, you must be sure to copy the PGA as well as the page itself (TIOATDL + 4 bytes in all).

7. The usual restriction against mixing text with mapped output in the same logical method does not apply here, because the page is already formed.

Chapter 39. Message routing

The message routing facilities of BMS allow you to send messages to terminals other than the principal facility of your task (your task does not even need to have a principal facility). Routing does not give your task direct control of these terminals, but instead causes the scheduling of a task for each destination to deliver your message. These tasks execute the CICS-supplied transaction CSPG, the same one used for delivery of messages with a disposition of PAGING to your own terminal. Thus the operator at a display terminal who receives a routed message uses CSPG requests to view the message. (See “Terminal operator paging: the CSPG transaction” on page 510 for more information about CSPG.)

Message routing is useful for message-switching and broadcasting applications, and also for printing (see “Using CICS printers” on page 415). It is the basis for the CICS-supplied transaction CMSG, with which terminal users can send messages to other terminals and users. See the *CICS Supplied Transactions* for an explanation of CMSG and what you can do with it.

To route a message, you start by issuing a ROUTE command. This command tells BMS where to send the message, when to deliver it, what to do about errors, and other details. Then you build your message. It can be a mapped or text message, but it must be a logical message (that is, either ACCUM or PAGING present), and the disposition must be either PAGING or SET, not TERMINAL. PAGING is the more common choice and is assumed in the discussion that follows. We explain SET in a routing context in “Routing with SET” on page 535.

Your ROUTE command is in effect until you end your message with a SEND PAGE command, and you must not issue another one until this occurs. (If you issue ROUTE while building your message you get an invalid request response; if you issue a second ROUTE before you start your logical message, it simply replaces the first one.) You can also terminate a message with PURGE MESSAGE, if you decide you do not want to send it. PURGE MESSAGE causes the routing environment established with your ROUTE command to be discarded, along with your logical message.

This chapter describes:

- “Message destinations”
- “Route list format” on page 530
- “Message delivery” on page 531
- “Recoverable messages” on page 532
- “Programming considerations with routing” on page 534

Message destinations

You can specify destinations for your routed message in three different ways:

- You can request that certain classes of operators receive the message, by using the OPCLASS option of the ROUTE command. Classes are associated with an operator in the RACF user definition or a CICS sign-on table entry.
- You can name particular operators who are to receive the message by using a route list, to which you point with the LIST option of the ROUTE command. Operators are identified by a 3-character OPIDENT value, which is also assigned in the RACF definition or a sign-on table entry.

- You can name particular terminals which are to receive the message; this is also done with a route list. Terminals are identified by their 4-character TERMID value, and, for terminal types to which they apply, a 2-character logical device code.

Note: If you need to know the identifier or operator class values for the operator signed on at your principal facility to specify the destination of your routed message, you can use the ASSIGN command with the OPID or OPCLASS options to find out.

Eligible terminals

To format a message properly for a particular destination, BMS needs to know the characteristics of the terminal for which it is formatting. This is true even for destinations that you designate by operator or operator class. The first step in processing a route list, therefore, is to translate your destinations to a list of *terminals* to which the message *may* be delivered. This “eligible terminal” list combines the information in your route list and your OPCLASS specification with the state of the terminal network *at the time of the ROUTE command*.

Later, when your message is ready for delivery, BMS uses this list to decide which terminals actually get your message. A terminal must be on the list to receive the message, but its presence there does not guarantee delivery. There may be operator restrictions associated with the terminal and, because delivery occurs later in time, the status or even the nature of the terminal may have changed.

Both at the time the list is built and at the time of delivery, BMS is restricted to the terminal definitions installed in its own CICS region (where the routing task is running, or ran) and may not have all of the terminal definitions you expect. First, terminals that are autoinstalled may not be logged on either at the time of the ROUTE, excluding them from inclusion on the list, or at the times sending is attempted, preventing delivery.

In a multiple-region environment, there is the additional possibility that terminals known to one region may not be known to another. (It depends on how they are defined, as explained in *CICS Application Programming Reference*.) In particular, if a terminal definition is shared among regions by designating it as SHIPPABLE in the region that owns it, the terminal is not defined in any other region until something occurs to cause shipment there. This usually happens the first time the terminal routes a transaction to the region in question. Consequently, a ROUTE in this region cannot include the terminal before the first such event occurs.

The following sections describe how BMS builds the list of eligible terminals. This occurs at the time of the ROUTE command:

Destinations specified with OPCLASS only

If you specified operator classes (the OPCLASS option) but no route list, BMS scans all the terminal definitions in the local system. Any terminal that meets all these conditions gets on the eligible terminal list:

- The terminal is of a type supported by BMS.
- The terminal can receive routed messages not specifically addressed to it (ROUTEDMSGs (ALL) in the terminal definition).
- An operator is signed on at the terminal.
- The operator belongs to one of the operator classes in your OPCLASS list.

The resulting entry is marked so that delivery occurs only when and if an operator belonging to at least one of the operator classes in your OPCLASS list is signed on. (This operator does not have to be the one that was signed on at ROUTE time.)

OPCLASS and LIST omitted

If you specify neither operator classes nor a route list, BMS puts every terminal that meets the first two tests above on the list, and sets no operator restrictions on delivery. In a network where many terminals are eligible to receive all routed messages, this is a choice you almost certainly want to avoid.

Route list provided

If you provide a route list, BMS builds its list from yours instead of scanning the terminal definitions. Each of your entries is processed as follows. Processing includes setting a status flag in the list entry to tell you whether the entry was used or skipped and why.

- If the entry contains a terminal identifier but no operator identifier, the terminal goes on the eligible list, provided it is defined, of a type supported by BMS, and eligible to receive routed messages. If BMS cannot find the terminal definition, it sets the “entry skipped” and “invalid terminal identifier” bits (X'C0') in the status flag of the route list entry; if the terminal exists but is not supported by BMS or is not allowed to receive any routed messages, the “entry skipped” and “terminal not supported under BMS” bits get set (X'A0').

Note: The eligibility of a terminal to receive routed messages is governed by the ROUTEDMSG option in the terminal definition. Three values are possible: a terminal may be allowed to receive all routed messages, only messages routed to it by terminal or operator name, or no routed messages at all. If you specified OPCLASS as well as a route list, BMS checks whether an operator belonging to one of the classes you listed is signed on at the terminal. If not, BMS sets the “operator not signed on” bit (X'10') in the status flag for the entry to inform you, but includes the terminal anyway. There are no operator restrictions associated with the list entry, even when you specify operator classes.

- If the entry contains both a terminal and an operator identifier, the terminal identifier is checked in the same way as it is without an operator identifier, and the same errors can occur. If the terminal passes these tests, it goes on the eligible list. However, the entry is marked such that the message can be delivered only when the operator named is signed on at the same terminal.

If this operator is not signed on to the terminal at the time of the ROUTE command, BMS notifies you by turning on the “operator not signed on” bit (X'10') in the status flag, but the terminal goes on the delivery list regardless of sign-on status. (OPCLASS is ignored entirely when an operator identifier is present.)

- If the entry contains only an operator identifier, BMS searches the terminal definitions until it finds one where the operator is signed on. (The operator may be signed on at additional terminals, but BMS ignores these.) If this terminal is of a type not supported by BMS, or if the terminal cannot receive routed messages, BMS sets the “entry skipped” and “operator signed on at unsupported terminal” bits (X'88') in the status flag. It also fills in the terminal identifier in your route list. If the terminal is suitable, BMS treats the entry as if you had specified both that terminal and operator identifier, as described above.

If the operator is not signed on anywhere, BMS sets the “entry skipped” and “operator not signed on” bits (X'90') in the status flag.

Route list format

BMS requires a fixed format for route lists. Each entry in the list is 16 bytes long, as follows:

Table 43. Standard route list entry format

Bytes	Contents
0-3	Terminal or logical unit identifier (four characters, including trailing blanks), or blanks
4,5	LDC mnemonic (two characters) for logical units with LDC support, or blanks
6-8	Operator identifier, or blanks
9	Status flag for the route entry
10-15	Reserved; must contain blanks

Either a terminal or an operator identifier must be present in each entry. A Logical Device Component(LDC) may accompany either; see “LDCs and routing” on page 550 for more information about LDCs.

The entries in the route list normally follow one another in sequence. However, they do not all have to be adjacent. If you have a discontinuity in your list, you end each group of successive entries except the last group with an 8-byte chain entry that points to the first entry in the next group. This entry looks like this:

Table 44. Route list chain entry format

Bytes	Contents
0,1	–2 in binary halfword format (X'FFFE')
2,3	Reserved
4-7	Address of the first entry in the next group of contiguous entries

The end of the entire list is signalled by a 2-byte entry containing a halfword value of –1 (X'FFFF').

Your list may consist of as many groups as you wish. There is an upper limit on the total number of destinations, but it depends on many variables; if you exceed it, BMS abends your task with abend code ABMC.

On return from a ROUTE command, BMS raises condition codes to signal errors in your list:

RTESOME

means that at least one of the entries in your route list could not be used and was skipped. The default action is to continue the routing operation, using the destinations that were processed successfully.

RTEFAIL

means that none of the destinations in your list could be used, and therefore no routing environment has been set up. The default action is to return control to your task. You should test for this condition, consequently, because with no routing environment, a subsequent BMS SEND command goes to the principal facility, which is probably not your intention.

In addition to the general information reflected by RTESOME and RTEFAIL, BMS tells you what it did with each entry in your list by setting the status flag (byte 9). A

null value (X'00') means that the entry was entirely correct. The high-order bit tells you whether the entry was used or skipped, and the other bits tell you exactly what happened. Here are the meanings of each bit being on:

ENTRY SKIPPED (X'80')

The entry was not used. When this bit is on, another bit is also on to indicate the reason.

INVALID TERMINAL IDENTIFIER (X'40')

There is no terminal definition for the terminal named in the entry. The entry is skipped.

TERMINAL NOT SUPPORTED UNDER BMS (X'20')

The terminal named in the route list entry is of a type not supported by BMS, or it is restricted from receiving routed messages. The entry is skipped.

OPERATOR NOT SIGNED ON (X'10')

The operator named in the entry is not signed on. Any of these conditions causes this flag to be set:

- Both an operator identifier and a terminal identifier were specified, and the operator was not signed on at the terminal. The entry is not skipped.
- An operator identifier was specified without a terminal identifier, and the operator was not signed on at any terminal. The entry is skipped.
- OPCLASS was specified on the ROUTE command, a terminal identifier was specified in the route list entry, and the operator signed on at the terminal did not have any of the specified operator classes. The entry is not skipped.

OPERATOR SIGNED ON AT UNSUPPORTED TERMINAL (X'08')

Only an operator identifier was specified in the route list entry, and that operator was signed on at a terminal not supported by BMS or not eligible to receive routed messages. The entry is skipped. The name of the terminal is returned in the terminal identifier field of the entry.

INVALID LDC MNEMONIC (X'04')

Either of these conditions causes this flag to be set:

- The LDC mnemonic specified in the route list is not defined for this terminal. That is, the terminal supports LDCs but it has no LDC list, or its LDC list is extended but does not contain this entry.
- The device type for this LDC entry is different from that of the first entry in the route list with an LDC (only one LDC device type is allowed, as explained in "LDCs and routing" on page 550).

The entry is skipped.

Note: CICS provides source code which defines a standard route list entry and the values you need to test status flag bit combinations. You can insert this code into your program with a COPY or INCLUDE of the member DFHURLDS, in the same way you can include the BMS attention identifier or attribute byte definitions.

Message delivery

We have just explained how BMS determines the terminals eligible to receive your routed message. Actual delivery occurs later in time, much later in some cases, depending on the scheduling options in your ROUTE command (INTERVAL, TIME, AFTER and AT). You can request delivery immediately, after an interval of time has elapsed, or at a particular time of day.

When the appointed time arrives, BMS attempts to deliver the message to every terminal on the eligible terminal list. All the following conditions must be met for the message to be delivered to any particular terminal:

- The terminal must be defined as a type supported by BMS, and the same type as when the ROUTE command was processed⁸. (Where there is a long delay between creation and delivery of a message, it is possible for the terminal defined with a particular TERMID to change characteristics or disappear, especially in an autoinstall environment.)
- The terminal must be in service and available (that is, there cannot be a task running with the terminal as its principal facility).
- The terminal must be eligible for automatic transaction initiation, or the terminal operator must request delivery of the message with the CSPG transaction.

Note: If several messages accumulate for delivery to a particular terminal, there is no guarantee that the operator will view them in any particular order. In fact, the CSPG transaction allows the operator to control delivery order in some situations. If a specific sequence of pages is required, you must send them as one message.

- If the delivery list entry restricts delivery to a particular operator or to operators in certain classes, the operator signed on at the terminal must qualify. (See “Message destinations” on page 527 for the OPCLASS and LIST specifications that produce these restrictions.)
- The purge delay must not have expired, as explained in the next section.

Undeliverable messages

If BMS cannot deliver a message to an eligible terminal, it continues to try periodically until one of the following conditions occurs:

- A change in terminal status allows the message to be sent.
- The message is deleted by the destination terminal operator.
- The purge delay elapses.

The purge delay is the period of time allowed for delivery of a message once it is scheduled for delivery. After this interval elapses, the message is discarded. The purge delay is a system-wide value, set by the PRGDLY option in the system initialization table. Its use is optional; if the systems programmer sets PRGDLY to zero, messages are kept indefinitely.

When BMS purges a message in this fashion, it sends an error message to the terminal you specify in ERRTERM. (If you use ERRTERM without a specific terminal name, it sends the message to the principal facility of the task that originally created the message. If you omit ERRTERM altogether, no message is sent.)

Recoverable messages

Between creation and delivery of a routed message with a disposition of PAGING, BMS stores the message in CICS temporary storage, just as it does in the case of an ordinary PAGING message. Consequently, you can make your routed messages recoverable by your choice of the REQID option value, just as in the case of a nonrouted message. (See “Logical message recovery” on page 511.)

8. A 3270 terminal need not have exactly the same extended attributes that it had at the time the ROUTE command was issued, because BMS removes unsupported attributes from the data stream at the time of delivery.

If you are routing to more than one type of terminal, BMS builds a separate logical message for each type, with the appropriate device-dependent data stream, and uses a separate temporary storage queue for each type.

Note: For terminal destinations that have the alternate screen size feature, where two message formats are possible, BMS chooses the default size if the profile under which the task creating the message specifies default size, and alternate size if the profile specifies alternate size.

All of the logical messages use the same REQID value, however, so that you can still choose whether they are recoverable or not.

BMS also uses temporary storage to store the list of terminals eligible to receive your message and to keep track of whether delivery has occurred. When all of the eligible terminals of a particular type have received a message, BMS deletes the associated logical message. When all of the destinations have received delivery, or the purge delay expires, BMS erases all of the information for the message, reporting the number of undeliverable messages by destination to the master terminal operator message queue.

Message identification

You can assign a title to your routed message if you wish. The title is not part of the message itself, but is included with the other information that BMS maintains about your message in CICS temporary storage. Titles are helpful in situations where a number of messages may accumulate for an operator or a terminal, because they allow the operator to control the order in which they are displayed. (See the “query” option of the CSPG command in the *CICS Supplied Transactions* manual.)

To assign a title, use the TITLE option of the ROUTE command to point to a data area that consists of the title preceded by a halfword binary length field. The length includes the 2-byte length field and has a maximum of 64, so the title itself may be up to 62 characters long. For example:

```
01 MSG-TITLE.
02 TITLE-LENGTH    PIC S9(4) COMP VALUE +19.
02 TITLE-TEXT      PIC X(17) VALUE 'MONTHLY INVENTORY'.
...
EXEC CICS ROUTE TITLE(MSG-TITLE)....
```

Figure 115. Assigning a title

Programming considerations with routing

For the most part, you build a routed message in the same way you do a nonrouted message. However, because BMS builds a separate logical message for each terminal type among your destinations, there are differences. The first involves page overflow.

Routing and page overflow

Because different types of terminals have different page capacities, page overflow may occur at different times for different types. If you are using SEND MAP commands and intercepting overflows, your program gets control when overflow occurs for each page of each logical message that BMS is creating in response to your ROUTE.

If you want to number your pages or do page-dependent processing at overflow time, you may need to keep track of information for each terminal type separately. Data areas kept for this purpose are called **overflow control areas**. You can tell how many such areas you need (that is, how many different terminal types appeared in your ROUTE command) by issuing an ASSIGN command with the DESTCOUNT option after your ROUTE and before any BMS command that could cause overflow. Issued at this time, ASSIGN DESTCOUNT returns a count of the logical messages that BMS builds.

When overflow occurs, you can use the same command to determine for which logical message overflow occurred. At this time ASSIGN DESTCOUNT returns the relative number of that message among those BMS is building for this ROUTE command. If you are using overflow control areas, this number tells you which one to use. If you use ASSIGN PAGENUM at this time, BMS returns the number of the page that overflowed as well.

To handle the complication of different overflow points for different terminal types, the processing you need to do on overflow in a routing environment is:

- Determine which logical message overflowed with ASSIGN DESTCOUNT (unless you are doing very simple overflow processing).
- Send your trailer maps for the current page, followed by headers for the next page, as you do in a non-routing environment (see “Page breaks: BMS overflow processing” on page 514). While the OVERFLOW condition is in force, these SEND MAP commands apply only to the logical message that overflowed (you would not want them in a logical message where you were mid-page, and BMS makes no assumptions about different terminal types that happen to have the same page capacity).
- Reissue the command that caused the overflow, as you do in a non-routing environment. After you do, however, you must retest for overflow and repeat the whole process, until overflow does not occur. This procedure ensures that you get the trailers and headers and the map that caused the overflow into each of the logical messages that you are building.

Routing with SET

When you specify a disposition of SET in a routing environment, no messages are sent to the destinations in your route list, because the pages are returned to your program as they are completed. However, the ROUTE command is processed in the usual way to determine these destinations and the terminal types among them. BMS builds a separate logical message for each type, as usual, and returns a page to the program each time one is completed for any of the terminal types. BMS raises the OVERFLOW and RETPAGE conditions as it does with a disposition of PAGING. Consequently, ROUTING with SET allows you to format messages for terminal types other than that of your principal facility.

Interleaving a conversation with message routing

While you are building a message to be routed, you can use BMS SEND commands as well as RECEIVE MAP and terminal control commands to converse with your principal facility. (Without routing, you cannot use BMS SENDs, as noted in “Building logical messages” on page 507.) Such SEND commands must have a disposition option of TERMINAL rather than PAGING or SET, and must not specify ACCUM. The associated inputs and outputs are processed directly and do not interfere with your logical message, even if your terminal is one of the destinations of the message.

Chapter 40. The MAPPINGDEV facility

Minimum BMS function assumes that the principal facility of your task is the **mapping device** that performs input and output mapping operations for the features and status that is defined in the TCTTE (Terminal Control Table entry).

The principal facility for transactions using BMS function should have a device type supported by BMS. However, the MAPPINGDEV facility is an extension of minimum BMS that allows you to perform mapping operations for a device that is **not** the principal facility. When the MAPPINGDEV request completes, the mapped data is returned to the application. BMS does not have any communication with the MAPPINGDEV device.

You can specify the MAPPINGDEV option on the RECEIVE MAP command (see the *CICS Application Programming Reference* manual) and the SEND MAP command, (see the *CICS Application Programming Reference* manual) but not on any other BMS command.

The TERMID specified in the MAPPINGDEV option must represent a device in the 3270 family supported by BMS. If the device is partitioned, it is assumed to be in base state. Outboard formatting is ignored.

Data is mapped in exactly the same way as for minimum BMS, and there is no need to change mapset definitions or to re-generate mapsets.

This chapter describes:

- “SEND MAP with the MAPPINGDEV option”
- “RECEIVE MAP with the MAPPINGDEV option” on page 538
- “Sample assembler MAPPINGDEV application” on page 539

SEND MAP with the MAPPINGDEV option

Your SEND MAP commands that have the MAPPINGDEV option must also specify the SET option. (The SET option provides BMS with a pointer that sets the address of the storage area that contains the mapped output datastream.)

If you have storage protection active, the data is returned in storage in the key specified in the TASKDATAKEY option of the transaction definition. The storage is located above or below the line depending on which TASKDATALOC option of the transaction definition you have specified.

The storage area is in task-related user storage but in the format of a TIOA (Terminal Input/Output Area). The application can reference the storage area using the DFHTIOA copybook. The TIOATDL field, at offset 8, contains the length of the datastream that starts at TIOADBA, at offset 12, in the storage area. The length value placed in TIOATDL does not include the length of the 4-byte page control area, which contains information such as the extended attributes that have been used in the datastream and can be referenced using the DFHPGADS copybook.

The storage area usually has a length greater than the datastream because the storage area is allocated before the exact length of the output datastream is determined. This storage area is in a form that can be used in a SEND TEXT MAPPED command.

If you are familiar with using the SET option without the MAPPINGDEV option, (see “Protection” on page 393 for details) you know that the datastream is returned to the application indirectly by a list of pages. However, when MAPPINGDEV is specified, a direct pointer to the storage area containing the datastream is returned to your application.

When the SEND MAP MAPPINGDEV command completes its processing, the storage area is under the control of the application and remains allocated until the end of the transaction unless your application FREEMAINs it. You are advised to FREEMAIN these storage areas, for long-running transactions but CICS frees these areas when the task terminates.

RECEIVE MAP with the MAPPINGDEV option

You must specify the FROM option when using the MAPPINGDEV option on the RECEIVE MAP command. BMS needs the FROM option to supply a formatted 3270 input datastream that is consistent with the datastream returned via a Terminal Control RECEIVE command (that is, a normal input 3270 datastream). The only difference is that it does not start with an AID and input cursor address because this information is removed from the input datastream by terminal control, but there are options on the RECEIVE MAP command that allow you to specify an AID value and input cursor position when the MAPPINGDEV option is specified. If the datastream contains an AID and input cursor address, they are ignored by BMS.

When neither option is specified, BMS assumes that the input data operation was terminated with the ENTER key, and returns the appropriate AID value to the application from the EIBAID field. BMS also assumes that the input cursor was positioned at the home address and returns a value of zero to the application from the EIBCPOSN field.

The new AID option of the RECEIVE MAP command allows your application to specify an AID value which, if specified, overrides the default value of ENTER. Whether provided by the application, or defaulted by BMS, the AID value that you established causes control to be passed, when applicable, to the routine registered by a previous HANDLE AID request issued by the application.

The new CURSOR option of the RECEIVE MAP command allows your application to specify an input cursor position which, if specified, overrides the default value of zero. Whether provided by the application, or defaulted by BMS, the input cursor value is used in cursor location processing when you define the map with CURSLOC=YES.

As with the minimum BMS RECEIVE MAP command, the mapped data is returned to your application by the INTO or SET option. If neither option is specified, the CICS translator attempts to apply a default INTO option by appending the character 'I' to the map name.

When you use the SET option with the MAPPINGDEV option, it must provide a pointer variable that BMS sets with the address of the storage area containing the mapped input datastream. The data is returned in task-related user storage. If storage protection is active, the data is returned in storage in the key specified by the TASKDATAKEY option of the transaction definition. The storage is located above or below the line depending on the TASKDATALOC option of your transaction definition.

When the RECEIVE MAP MAPPINGDEV command completes its processing successfully, the storage area is returned by the SET option and is under the control of the application and remains allocated until the end of the transaction unless your application FREEMAINS it. You are advised to FREEMAIN these storage areas, for long-running transactions but CICS frees these areas when the task terminates.

Sample assembler MAPPINGDEV application

Figure 116 is a modification of the FILEA operator instruction sample program, and uses the same mapset named DFH\$AGA.

This application is only intended to demonstrate how to code the keywords associated with the MAPPINGDEV facility, and as a means of testing this function. It is not offered as a recommended design for applications that make use of the MAPPINGDEV facility.

```

DFH$AMNX CSECT
*
      DFHREGS
DFHEISTG DSECT
OUTAREA DS    0CL512
          DS    CL8
OUTLEN   DS    H
          DS    H
OUTDATA  DS    CL500
INLEN    DS    H
INAREA   DS    CL256
PROOF    DS    CL60
          COPY  DFH$AGA
          COPY  DFHBMSCA
DFH$AMNU CSECT
      EXEC CICS HANDLE AID PF3(PF3_ROUTINE)
*
      XC      DFH$AGAS(DFH$AGAL),DFH$AGAS
      MVC     MSGO(L'APPLMSG),APPLMSG
      EXEC    CICS SEND MAP('DFH$AGA') FROM(DFH$AGAO) ERASE
              MAPPINGDEV(EIBTRMID) SET(R6)
      MVC     OUTAREA(256),0(R6)
      MVC     OUTAREA+256(256),256(R6)
      EXEC    CICS SEND TEXT MAPPED FROM(OUTDATA) LENGTH(OUTLEN)
*
      EXEC    CICS RECEIVE INTO(INAREA) LENGTH(INLEN)
              MAXLENGTH(MAXLEN)
*
      EXEC    CICS RECEIVE MAP('DFH$AGA') SET(R7) LENGTH(INLEN)
              MAPPINGDEV(EIBTRMID) FROM(INAREA)
              CURSOR(820) AID(=C'3')
*
      XC      PROOF,PROOF
      MVC     PROOF(25),=C'You just keyed in number '
      MVC     PROOF+25(6),KEY1-DFH$AGAI(R7)

```

Figure 116. ASM example of a MAPPINGDEV application (Part 1 of 2)

```

FINISH    DS      0H
          EXEC CICS SEND TEXT FROM(PROOF) LENGTH(60) ERASE FREEKB
          TM      MSGF-DFH$AGAI(R7),X'02'
          BNO     RETURN
          XC      PROOF,PROOF
          MVC      PROOF(33),=C'Input cursor located in MSG field'
          EXEC CICS SEND TEXT FROM(PROOF) LENGTH(60) ERASE FREEKB
*
*         THE RETURN COMMAND ENDS THE PROGRAM.
*
RETURN    DS      0H
          EXEC CICS RETURN
*
PF3_ROUTINE DS 0H
          XC      PROOF,PROOF
          MVC      PROOF(30),=C'RECEIVE MAP specified AID(PF3)'
          B        FINISH
MAXLEN    DC      H'256'
APPLMSG    DC      C'This is a MAPPINGDEV application'
          END

```

Figure 116. ASM example of a MAPPINGDEV application (Part 2 of 2)

Chapter 41. Partition support

Partitions are the first of several special hardware features that BMS supports. **Standard** BMS is required for partitions.

Some IBM displays allow you to divide the screen into areas which you can write to and read from separately, as if they were independent screens. The areas are called partitions, and features of BMS that allow you to take advantage of the special hardware are collectively called “partition support”.

The IBM 3290 display, which is a member of the 3270 family, and the IBM 8775 are the primary examples of devices that support partitioning. You should consult the device manuals⁹ to understand the full capabilities of a partitioned device, but the essential features are these:

- You can divide the physical screen into any arrangement of one to eight non-overlapping rectangular areas. The areas are independent from one other, in the sense that the operator can clear them separately, the state of the keyboard (locked or unlocked) is maintained separately for each, and you write to and read from them one at a time.
- Only one partition is **active** at any given time. This is the one containing the cursor. The operator is restricted to keying into this partition, and the cursor wraps at partition boundaries. When a key that transmits data is depressed (the ENTER key or one of the program function keys), data is transmitted only from the active partition.
- The operator can change the active partition at any time by using the “jump” key; your program can also, as explained in “Determining the active partition” on page 545.
- BMS also writes to only one partition on a given SEND, but you can issue multiple SENDs and you do not have to write to the active partition.
- The partition configuration is sent to the device as a data stream, so that you can change the partitions for each new task or even within a task. The BMS construct that defines the partitions is called a **partition set** and is described in “Partition definition” on page 543.
- You also can use the terminal in **base state** (without partitions) and you can switch from partitioned to base state with the same command that you use to change partition arrangements.
- When you specify how to partition the screen area, you also divide up the hardware buffer space from which the screen is driven. In partitioned devices, the capacity of the buffer is generally greater than that of the screen, so that some partitions can be assigned extra buffer space. The screen area allocated to a partition is called its **viewport** and the buffer storage is called its **presentation space**.

BMS uses the presentation space as its page size for the partition, so that you can send as much data as fits there, even though not all of it can be on display at once. Keys on the device allow the operator to scroll the viewport of the partition vertically to view the entire presentation space. Scrolling occurs without any intervention from the host.

- Some partitioned devices allow you to choose among character sets of different sizes. We talk about this in “3290 character size” on page 544.

9. *IBM 3290 Information Display Panel Description and Reference for the 3290 and IBM 8775 Display Terminal Component Description for the 8775.*

In spite of the independence of the partitions, the display is still a single terminal to CICS. You cannot have more than one task at a time with the terminal as its principal facility, although you can use the screen space cooperatively among several pseudoconversational transaction sequences if they use the same partition set (see “Terminal sharing” on page 548).

Note: The 3290 can be configured internally so that it behaves as more than one logical unit (to CICS or any other system); this definition is separate from the partitioning that may occur at any one of those logical terminals.

This chapter describes:

- “Uses for partitioned screens”
- “Partition definition” on page 543
- “Establishing partitioning” on page 544
- “Partition options for BMS SEND commands” on page 545
- “Partition options for BMS RECEIVE commands” on page 546
- “Partitions and logical messages” on page 546
- “Attention identifiers and exception conditions” on page 547
- “Terminal sharing” on page 548

Uses for partitioned screens

Partitioned screens are particularly useful in certain types of application. For example:

Scrolling

For transactions that produce more output than fits on a single screen, scrolling is an alternative to BMS terminal paging (see “Output disposition options: TERMINAL, SET, and PAGING” on page 489). For example, you can define a partition set that consists of just one partition, where the viewport is the whole screen and the presentation space is the entire buffer. You can write to the entire buffer as a single page, and the operator can scroll through the data using the terminal facilities. Response time to scrolling requests is very short, because there is no interaction with the host. You are limited to the capacity of the buffer, of course.

You may also want to scroll just part of the screen and use some partitions for fixed data.

Data entry

Another good use for a partitioned screen is “heads down” data entry, where the operator’s productivity depends on how fast the application can process an input and reopen the keyboard for the next. With a partitioned screen, you can divide the screen into two identical entry screens. The operator fills one, presses Enter, and then fills the second one while the data entry transaction is processing the first input. If the input is good, the program simply erases it in preparation for the next entry; if not, there is still an opportunity for the operator to make corrections without losing subsequent work. The *CICS 4.1 Sample Applications Guide* contains an example of such a data entry transaction.

Lookaside

In many online operations, the operator sometimes needs to execute a second transaction in order to finish one in progress. Order entry is an example, where the operator may have to look up codes or prices to complete an entry. Many inquiries are similar. The initial inquiry brings back a summary list of hits. The operator

selects one and asks for further detail, then may need to select another for detail, and so on. In such cases, a partitioned screen allows the operator to do the second task while keeping the output of the first, which is needed later, on the screen. The *CICS 4.1 Sample Applications Guide* also contains an example of a lookaside transaction.

“Help” text is still another example of “lookaside”. If you allocate one partition of the screen to this text, the operator can get the required tutorial information without losing the main screen.

Data comparison

Applications in which the operator needs to compare two or more sets of data simultaneously are also excellent candidates for a partitioned screen. Partitioning allows a side-by-side comparison, and the scrolling feature makes it possible to compare relatively large documents or records.

Error messages

If you partition a screen and allocate one area to error messages and other explanatory text, usability is enhanced because the operator always knows where to look for messages, and the main screen areas are never overwritten with such information. CICS sends its own messages to such a partition if you designate one in your partition set, as we explain in “Partition definition.”

Partition definition

Each partitioning of a screen is defined by a partition set, which is a collection of screen areas (partitions) intended for display together on a screen. You define a partition set with assembler macros, just as you do map sets. There are two of them: DFHPSD and DFHPDI.

The partition set definition begins with a DFHPSD (partition set definition) macro, which defines:

- The name of the partition set
- Screen size (BMS makes sure that the partition viewports do not exceed the total space available)
- Default character cell size (we talk about cell size in “3290 character size” on page 544)
- The partition set suffix, used to associate the partition set with a particular screen size (see “Establishing partitioning” on page 544)

After the initial DFHPSD macro, you define each partition (screen area) with a DFHPDI macro. DFHPDI specifies:

- The identifier of the partition within the partition set.
- Where the partition is located on the screen.
- Its viewport size (in lines and columns).
- The presentation space associated with the viewport (that is, the amount of buffer space allocated), also in lines and columns. Because scrolling is strictly vertical, BMS requires that the width of the presentation space match the width of the viewport.
- The character size to be used.
- The map set suffix associated with the partition, used to select the map set appropriate for the partition size.

- Whether the partition may receive CICS error messages (BMS sends certain error messages that it generates to a partition so designated, if there is one).

You end the partition set with a second DFHPSD macro, containing only the option TYPE=FINAL. See the *CICS Application Programming Reference* manual.

Because these are assembler macros, you need to follow assembler format rules in creating them. See “Writing BMS macros” on page 463 if you are not familiar with assembler language. After you do, you need to assemble and link-edit your partition set. The resulting load module can be placed in the same library as your map sets, or in a separate library if your installation prefers. Your systems staff also need to define each partition set to the system with a PARTITION definition.

3290 character size

The 3290 hardware allows you to use up to eight different character sets, of different sizes. Two sets come with the hardware; the others can be loaded with a terminal control SEND command. (Refer to the *IBM 3290 Information Display Panel Description and Reference* manual for details.)

Each character occupies a rectangular **cell** on the screen. Cell size determines how many lines and columns fit on the screen, or in a particular partition of the screen, because you can specify cell size by partition. Cells are measured in pels (picture elements), both vertically and horizontally. The smallest cell allowed is 12 vertical pels by 6 horizontal pels. The 3290 screen is 750 pels high and 960 pels wide. Using the minimum cell size, therefore, you can fit 62 characters vertically (that is, have 62 lines), and 160 characters horizontally (for 160 columns). (The 3290 always selects the character set that best fits your cell size, and places the character at the top left corner of the cell.)

Partition sizes are expressed in lines and columns, based on the cell size you specify for the partition, which is expressed in pels. (The name of the option is CHARSIZE, but it is really cell size.) To make sure your partitions fit on the screen, you need to work out your allocation in pels, although BMS tells you when you assemble if your partitions overlap or does not fit on the screen. The partition height is the product of the number of rows in the partition and the vertical CHARSIZE dimension; the partition width is the product of the number of columns and the horizontal CHARSIZE value.

If you do not specify a CHARSIZE size in your DFHPDI partition definition, BMS uses the default given in the DFHPSD partition set definition. If DFHPSD does not specify CHARSIZE either, BMS uses the default established for the terminal when it was installed. If you specify cell size for some but not all partitions, you must specify a default for the partition set too, so that you do not mix your choices with the installation default.

Establishing partitioning

You can tell BMS which partition set to load for a particular transaction by naming it in the PARTITIONSET option of the TRANSACTION definition. If you do this, and the named partition set is not already loaded at the terminal, BMS adds the partition definitions to your data on the first BMS SEND in the task.

You can also direct BMS not to change the partitions from their current state (PARTITIONSET=KEEP in the TRANSACTION definition) or indicate that you load

the partitions yourself (PARTITIONSET=OWN). If you do not specify any PARTITIONSET value, BMS sets the terminal to base state (no partitions) at the time it initiates the transaction.

Whatever the PARTITIONSET value associated with the transaction, a task can establish new partitions at almost any time with a SEND PARTNSET command, except that you cannot issue the command while you are building a logical message.

SEND PARTNSET does not send anything to the terminal immediately. Instead, BMS remembers to send the partition information along with the next BMS command that sends data or control information, just as it sends a partition set named in the PARTITIONSET option of the TRANSACTION definition on the first BMS SEND. Consequently, you must issue a SEND MAP, SEND TEXT or SEND CONTROL command before you issue a RECEIVE or RECEIVE MAP that depends on the new partitions.

Note: You can get an unexpected change of partitions in the following situation. If CICS needs to send an error message to your terminal, and the current partition set does not include an error partition, CICS returns the terminal to base state, clear the screen, and write the message. For this reason, it is a good idea to designate one partition as eligible for error messages in every partition set.

When BMS loads a partition set, it suffixes the name requested with the letter that represents your terminal type if device-dependent support is in effect, in order to load the one appropriate to your terminal. It takes suffix from the ALTSUFFIX option value of the TYPETERM definition associated with your terminal. Partition set suffixing is analogous to map set suffixing, and the same sequence of steps is taken if there is no partition set with the right suffix (see “Device-dependent maps” on page 472).

Partition options for BMS SEND commands

As noted earlier, when you write to a partitioned screen, you write to only one partition, and the effects of your command are limited to that partition. ERASE and ERASEAUP clear only within the partition, and FREEKB unlocks the keyboard only when the partition becomes active.

You can specify the partition to which you are sending with either the PARTN option in your map definition or with the OUTPARTN option on your SEND MAP. OUTPARTN overrides PARTN. If you don't specify either, BMS chooses the first partition in the set.

The use of partitions affects the suffixing of map set names that we described in “Device-dependent maps” on page 472. The map set suffix is taken from the MAPSFX value for the partition instead of being determined as described in that section.

Determining the active partition

When you send to a partition, you can move the cursor to that partition or another one. A value of ACTIVATE in the PARTN option of the map definition puts the cursor in the partition to which you are writing. If you specify ACTPARTN on your BMS SEND command, you can name any partition (not necessarily the one to which you are writing), and you override the ACTIVATE specification. Both

ACTIVATE and ACTPARTN unlock the keyboard for the active partition, as well as placing the cursor there. If neither is present, the cursor does not move and the keyboard is not unlocked.

Although you can make a partition active by placing the cursor there when you send, you do not have the last word on this subject, because the operator can use the jump key on the terminal to move the cursor to another partition. This can complicate receiving data back from the terminal, but BMS provides help, as we are about to explain.

Partition options for BMS RECEIVE commands

When you issue a RECEIVE MAP command, you can tell BMS from which partition you expect data (that is, which partition you expect to be active) with either the PARTN option in the map definition or with the INPARTN option on your RECEIVE MAP. INPARTN overrides PARTN. If you do, and the operator transmits from a different partition than the one you named, BMS repositions the cursor in the partition you named, unlocks the keyboard and repeats the RECEIVE command. It also sends a message to the error partition (the one with ATTRB=ERROR) asking the operator to use the right partition. (No message is sent if there is no error partition.) The input from the wrong partition is discarded, although it is not lost, because it can be reread later. BMS does this up to three times; if the operator persists for a fourth round, BMS raises the PARTNFAIL condition.

You do not have to specify an input partition; sometimes there is only one that allows input, and sometimes the same map applies to all. If you issue RECEIVE MAP without INPARTN and there is no PARTN option in the map, BMS accepts data from any partition and map it with the map named in the command. You also can determine the partition afterward, if you need to, with an ASSIGN command containing the INPARTN option.

INPARTN is not set until after the first BMS operation, however, and so if you need to know which partition is sending to select the right map, you need another approach. In this situation, you can issue a RECEIVE PARTN command, which reads data unmapped and tells you which partition sent it. Then you issue a RECEIVE MAP command using the map that matches the partition with the FROM option, using the map that matches the partition. RECEIVE MAP with FROM maps data already read, as explained in “Formatting other input” on page 505.

ASSIGN options for partitions

In addition to the INPARTN option just described, there are three other ASSIGN options to help you in programming for a partitioned terminal. The PARTNS option tells you whether the terminal associated with your task supports partitions, and the PARTNSET option returns the name of the current partition set (blanks if none has been established). The fourth ASSIGN option, PARTNPAGE applies only to logical messages, which we talk about in “Partitions and logical messages.”

Partitions and logical messages

When you build a BMS logical message for a terminal for which partitions have been established, you can direct the pages of the message to multiple partitions. You can even send text output to some partitions and mapped output to others, provided you do not mix them in the same partition. (This is an exception to the normal rule against mixing text and mapped output in a logical message.)

When the output is displayed, the first page for each partition is displayed initially. The pages are numbered by partition, and CSPG commands that the operator enters into a particular partition apply only to that partition, with the exception of the page purge command. The purge command deletes the entire logical message from all partitions.

On each BMS SEND that contributes to the message, you specify the partition to which the output goes. If you are not using ACCUM, BMS builds a page for that partition. If you are using ACCUM, BMS puts the output on the current page for that partition. Page overflows therefore occur by partition. If you are intercepting overflows and are not sure in which partition the overflow occurred, you can use the PARTNPAGE option of the ASSIGN command to find out.

Note: Because BMS uses both the page size and partition identifiers in building a logical message, you cannot change the partitions mid-message.

The bookkeeping required to handle page overflow when you are distributing pages among partitions is analogous to that required in a routing environment (see “Routing and page overflow” on page 534). In particular, you need to ensure that you finish overflow processing for one partition before doing anything that might cause overflow in another. Failure to do so can cause program loops as well as incorrect output.

Partitions and routing

You cannot route a logical message written to multiple partitions. BMS ignores the OUTPARTN and ACTPARTN options on BMS SEND commands in a routing environment.

You can route an ordinary message to a terminal that supports partitions, but BMS builds the message and the CSPG transaction displays it using the terminal in base (unpartitioned) state.

You also cannot use partitions and logical device codes together (LDCs are described in “Logical device components” on page 549). In addition, you cannot use partitions in combination with GDDM, although you can use partitions with outboard formats (see “Outboard formatting” on page 554).

Attention identifiers and exception conditions

Partitioned terminals have a CLEAR PARTITION key that clears the active partition in the same way that the CLEAR key clears the whole screen (CLEAR still does this on a partitioned terminal). You may need to check for this additional attention identifier in your program logic. The CLEAR PARTITION AID value is included in DFHAID (see “Using the attention identifier” on page 499).

There are also some new exception conditions associated with partitions, and new ways to get some of the old ones. The new ones include INVPARTN (naming a partition that does not exist in the partition set), INVPARTNSET (naming a module that is not a partition set), and PARTNFAIL (receiving from a partition other than the one the operator transmitted from). They are all described in the *CICS Application Programming Reference* manual with the commands to which they apply.

Terminal sharing

With proper planning, you can share a terminal among several processes by assigning each a separate partition. You cannot have more than one task in progress at once at a terminal, of course, but you can interleave the component tasks of several pseudoconversational transaction sequences at a partitioned terminal.

To take a very simple example, suppose you decide to improve response time for an existing pseudoconversational data entry transaction by letting the operator enter data in two partitions (see “Data entry” on page 542). You could modify the application to work on two records at once, or you could simply modify it to send to the same partition from which it got its input. Then you could run it independently from each partition.

You can establish the partitions with the `PARTITIONSET` option in the `TRANSACTION` definition (all of the transactions involved, if there are several in the sequence). As noted earlier, BMS does not reload the partitions as long as each transaction has the same `PARTITIONSET` value. Alternatively, you could establish the partitions with a preliminary transaction (for example, one that displayed the first entry screen in both partitions) and use a `PARTITIONSET` value of `KEEP` for the data entry transactions. Whenever you share a partitioned screen, whether among like transactions or different ones, you need to ensure that one does not destroy the partition set required by another. Also, if two different CICS systems may share the same screen, they should name partition sets in common, so that BMS does not reload the partitions when it should not.

If the hypothetical data entry transaction sequence uses the `TRANSID` option on the `RETURN` command to specify the next transaction identifier, you would need to make another small change, because the option applies to the whole terminal, not the partition. One solution would be to place the next transaction identifier in the first field on the screen (turning on the modified data tag in the field definition) and remove the `TRANSID` from the `RETURN`. CICS would then determine the next transaction from the input, as described in “How tasks are started” on page 142.

Chapter 42. Support for special hardware

In addition to partitions, BMS provides support these other special hardware features:

- Logical device components
- 10/63 magnetic slot reader
- Field selection features: cursor select, light pen, trigger fields
- Outboard formatting

The magnetic slot reader and outboard formatting both require **standard** BMS. Support for the cursor select key, light pen and trigger fields is included in **minimum**.

This chapter describes:

- “Logical device components”
- “10/63 magnetic slot reader” on page 551
- “Field selection features” on page 551
- “Cursor and pen-detectable fields” on page 552
- “Outboard formatting” on page 554

Logical device components

Logical device components (LDCs) are another special hardware feature supported by BMS. Like partitions, LDCs require **standard** BMS.

A terminal that supports LDCs is one that consists of multiple functional components (logical devices) controlled through a single point (the logical unit). The components might be a printer, reader, keyboard and display, representing a remote work station, or they might be multiple like devices, such as word processing stations or passbook printers. The IBM 3601 logical unit, the 3770 batch logical unit, 3770, and 3790 batch data interchange logical units, and LU type 4 logical units all support logical device components.

Because the logical unit is a single entity to CICS, but consists of components that can be written and read independently, the CICS application programming interface for LDC terminals looks similar to that for partitioned terminals, each LDC corresponding to one partition in a partition set. There are many differences, of course, and you should consult the CICS manual that describes CICS support for your particular terminal type. The sections which follow describe the major differences that affect programming, which are: .

- LDC definition
- SEND command options
- Logical messages
- Routing

Defining logical device components

The logical device components for a terminal are defined by a list called an LDC table. The TYPETERM component of the TERMINAL definition points to the table, which may be individual to the logical unit or shared by several logical units that have the same components. The table itself is defined with DFHTCT TYPE=LDC (terminal control) macros. (See *CICS Resource Definition Guide* for descriptions of both TYPETERM and the DFHTCT macros.)

An LDC table contains the following information for each logical device component of the logical unit:

- A 2-character logical device identifier. These identifiers are usually standard abbreviations, such as CO for console and MS for a magnetic stripe encoder, but they need not be.
- A 1-character device code, indicating the device type (console, card reader, word processing station). Codes are assigned by CICS from the device type and other information provided in the macro.
- A BMS page size. BMS uses this size, rather than one associated with the logical unit, because different logical devices have different page sizes.
- A BMS page status (AUTOPAGE or NOAUTOPAGE); see “The AUTOPAGE option” on page 510.

Sending data to a logical device component

You direct BMS output to a specific logical device component of a terminal by naming it in the LDC option of your SEND MAP, SEND TEXT, or SEND CONTROL command or the LDC option of your mapset. A value in the command overrides one in the map set. If the LDC does not appear in either place, BMS uses a default that varies with the terminal type.

LDCs and logical messages

When you build a BMS logical message for your own terminal, you can distribute pages of the message among different logical device components in the same way that you can direct pages to a logical message to different partitions. BMS accumulates pages separately for each logical device component in the same way that it does for partitions (see “Partitions and logical messages” on page 546). You can include both text and mapped output in the message, provided you do not send both to one LDC. Page overflow occurs by LDC, and terminal operator paging commands operate on a logical device component basis.

When retrieving pages, the operator (or user code in the device controller) must indicate the LDC to which the request applies, because not all devices have keyboards. As in the case of partitions, a message purge request deletes the entire message, from all LDCs. See the *CICS Supplied Transactions* manual for more detail on page retrieval for logical devices.

If you are intercepting page overflows, you can tell which LDC overflowed by issuing an ASSIGN command with either the LDCMNEM or LDCNUM option. Both identify the device which overflowed, the first by its 2-character name and the second by the 1-byte numeric identifier. You can determine the page number for the overflowing device with ASSIGN PAGENUM, just as with a partitioned device.

There is one restriction associated with LDCs and page overflow that is unique to LDCs. After overflow occurs, you must send both a trailer map for the current page and a header for the next one to the LDC that overflowed. BMS raises the INVREQ (invalid request) condition if you fail to do this.

LDCs and routing

Routing is supported in an LDC environment, provided the message goes to the same component type for every destination that supports LDCs. (You cannot route a multiple-LDC message.)

You can supply the LDC value in several ways:

- If you use the LDC option on your ROUTE command, the value supplied overrides all other sources and is used for all eligible destinations to which LDCs apply.
- If you specify an LDC in a route list entry (and not in the ROUTE command), that value is used for the associated destination. (If you specify both and they do not agree, the ROUTE list value is used and the discrepancy is flagged in the status flag of the entry.)
- If you specify neither, the value is determined from terminal and system LDC tables in the same way as it is in a non-routing environment when you omit the LDC from the BMS SEND command. (The value on the SEND command is ignored when routing is in effect.)

10/63 magnetic slot reader

Some IBM display terminals support a magnetic slot reader (MSR), a device that reads data from small magnetic cards, as an optional feature. The MSR has indicator lights and an audible alarm to prompt operator actions. Some terminals control the MSR themselves, but others, such as the IBM 8775 and the IBM 3643, let you control the functions of the reader by program.

CICS provides an ASSIGN command option, MSR, that tells you whether the principal facility of your task has an MSR or not.

With BMS, you can control the state of such an MSR by using the MSR option of the BMS SEND commands. This option transmits four bytes of control data to the attached MSR, in addition to display data sent to the terminal. BMS provides a copybook, DFHMSRCA, containing most of the control sequences you might need. The *CICS Application Programming Reference* manual describes the supplied constants and explains the structure of the control data, so that you can expand the list if you need to.

The control sequence that you send to an MSR affects the next input from the device; hence it has no effect until a RECEIVE command is issued. Input from MSRs is placed in the device buffer and transmitted in the same way as keyboard input. If the MSR input causes transmission, you can detect this by looking at the attention identifier in EIBAID. A value of X'E6' indicates input from the MSR, and X'E7' signals input from the MSR extended (a second MSR that may be present). See the *IBM 3270 Information Display System Data Stream Programmer's Reference* manual for information on how to format a screen for MSR input and other details on these devices.

Field selection features

BMS supports several special hardware features that allow the operator to enter and transmit input by selecting a field on the screen:

- Trigger fields
- Cursor selectable fields
- Light pen detection

Trigger field support

Trigger fields are a special hardware feature of certain types of terminal, such as the 8775. A field defined as a trigger field causes the terminal to transmit its contents if the operator moves the cursor out of the field when it is **primed**. The field gets primed when the operator moves the cursor into it and enters data or uses either the DELETE or ERASE EOF keys. It becomes unprimed after it causes

transmission, or if the operator uses the ERASE INPUT key, or after a send to the terminal (if you are using partitions, the send must be to the partition that contains the trigger field to have this effect).

You define a field as a trigger field by setting the VALIDN extended attribute to a value of TRIGGER, either in the map or by program override.

Only the field itself is sent when a trigger field causes transmission; other fields are not sent, even if they have been modified. You can detect a transmission caused by a trigger field by checking the attention identifier, which has a value of X'7F'.

Terminals that support the validation feature buffer the keyboard, so that the operator can continue to enter data while the host is processing an earlier transmission. The program processing such inputs needs to respond quickly, so that the operator does not exceed the buffer capacity or enter a lot of data before an earlier error is diagnosed.

The customary procedure is for the program receiving the input to check the contents of the trigger field immediately. If correct, the program simply unlocks the keyboard to let the operator continue (a BMS SEND command containing the FREEKB option does this). If the field is in error, you may wish to discard the stored keystrokes, in addition to sending a diagnostic message. Any of the following actions does this:

- A BMS SEND command that contains ERASE, ERASEAUP, or ACTPARTN or that lacks FREEKB
- A BMS SEND directed to a partition other than the one containing the trigger field (where partitions are in use)
- A RECEIVE MAP, RECEIVE PARTITION or terminal control RECEIVE command
- Ending the task

See the *IBM 3270 Information Display System Data Stream Programmer's Reference* manual for more information about trigger fields.

Cursor and pen-detectable fields

BMS also supports **detectable** fields, another special hardware feature available on some terminals. There are two hardware mechanisms for detectable fields: the "cursor select" key and the light pen. A terminal has either the key or a pen, not both. Both work the same way and, as the key succeeded the pen, we talk about the key.

For a field to be detectable, it must have certain field attributes, and the first character of the data, known as the **designator character**, must contain one of five particular values. You can have other display data after the designator character if you wish.

The bits in the field attributes byte that govern detectability also control brightness. High intensity (ATTRB=BRT) fields are detectable if the designator character is one of the detectable values. Normal intensity fields may or may not be detectable; you have to specify ATTRB=DET to make them so; nondisplay (ATTRB=DRK) fields cannot be detectable.

As usual, you can specify attributes and designator characters either in the map definition or by program override. However, DET has a special effect when it appears in an input-only map, as we explain in a moment.

Note that because high-intensity fields have, by definition, the correct field attributes for detectability, the terminal operator can make an *unprotected* high-intensity field detectable by keying a designator character into the first position of the field.

Selection fields

There are two types of detectable field, **selection** and **attention** fields; the type is governed by the designator character. A selection field is defined by a designator character of either a question mark (?) or a greater-than sign (>). The convention is that (?) means the operator has not selected whatever the field represents, and (>) means he has. The hardware is designed around this convention, but it is not enforced, and you can use another if it suits. You can initialize the designator to either value and initialize the modified data tag off or on with either value.

Every time the operator presses the cursor select key when the cursor is in a selection field, the designator switches from one value to the other (? changes to > and > changes to ?). The MDT is turned *on* when the designator goes from ? to > and *off* when the designator goes from > to ?, regardless of its previous state. This allows the operator to change his mind about a field he has selected (by pressing cursor select under it again) and gives him ultimate control over the status of the MDT. The MDT governs whether the field is included when transmission occurs, as it does for other fields. No transmission occurs at this time, however; selection fields do not of themselves cause transmission; that is the purpose of attention fields.

Attention fields

Attention fields are defined by a designator character of blank, null,¹⁰ or ampersand. In contrast to a selection field, when the cursor select key is pressed with the cursor in an attention field, transmission occurs.

If the designator character is an ampersand, the effect of pressing the cursor select key is the same as depressing the ENTER key. However, if the designator is blank or null, what gets transmitted is the address of every field with the MDT on, the position of the cursor, and an attention identifier of X'7E'. The *contents* of these fields are not transmitted, as they are with the ENTER key (or a cursor select with an ampersand designator). In either case, the fields with the MDT bit on may be selection fields or normal fields which the operator changed or which were sent with the MDT on.

BMS input from detectable fields

After transmission caused by a cursor-select attention field with a blank or null designator, BMS tells you which fields were transmitted (that is, had the MDT on) by setting the first position of the corresponding input (I) subfield to X'FF'. The first position is otherwise set to X'00'. You can tell which attention field caused transmission from this value if it was the only one transmitted, or from the position of the cursor otherwise.

#

If transmission is caused by a cursor-select attention field with an ampersand designator (or by the ENTER key or a PF key), the I subfield contains the contents of the field if the MDT is on and the L subfield reflects its length, as usual, except if the DET attribute is specified for a field in the input map (that is MODE=IN or MODE=INOUT, DATA=FIELD). After a RECEIVE MAP naming such a map, this I

10. A null in the data stream has the same effect as a blank in this function, but in BMS you should use a blank, because BMS does not transmit nulls in some circumstances, and because you cannot override the first position of a field with a null (see "Where the values come from" on page 486).

subfield contains X'FF' with a length of 1 if the field is selected (that is, if the MDT
was on), and a null (X'00') if not. BMS supplies no other input for the field, even if
some was transmitted.

Consequently, if you need to receive data from a detectable field as well as
knowing whether it was selected or not, you need to avoid the use of DET in an
input map. You can use separate maps for output and input and specify the DET
attribute only in the output map, or you can set the DET attribute in the datastream
sent by the program rather than in the map. For high intensity fields you do not
need to specify DET, because BRT implies DET. BMS will return the data for fields
specified as BRT in the input map.

You also need to ensure that the data gets transmitted. When the cause of transmission is the ENTER key, a PF key, or an attention field with an ampersand designator character, field data gets transmitted. It does not when the cause is an attention field with a blank or null designator.

See the *IBM 3270 Information Display System Data Stream Programmer's Reference* manual for more information about detectable fields.

Outboard formatting

Outboard formatting is a technique for reducing the amount of line traffic between the host processor and an attached subsystem. The reduction is achieved by sending only variable data across the network. This data is combined with constant data, such as a physical map, by a program within the subsystem. The formatted data can then be displayed.

You can use outboard formatting with a 3650 Host Communication Logical Unit, an 8100 Series processor with DPPX and DPS Version 2, or a terminal attached through a 3174 control unit. Maps used by the 3650 must be redefined using the 3650 transformation definition language before they can be used. For more information, see the section describing BMS in the *IBM CICS/OS/VS 3650/3680 Guide*. Maps to be used with the 8100 must be generated on the 8100 using either an SDF II utility or the interactive map definition component of the DPS Version 2.

If a program in the host processor sends a lot of mapped data to subsystems, you can reduce line traffic by telling BMS to transmit only the variable data in maps. The subsystem must then perform the mapping operation when it receives the data. BMS prefixes the variable data with information that identifies the subsystem map to be used to format the data.

Terminals that support outboard formatting have OBFORMAT(YES) in their TYPETERM definition. When a program issues a SEND MAP command for such a terminal, and the specified map definition contains OBFMT=YES, BMS assumes that the subsystem is going to format the data and generates an appropriate data stream. If you send a map that has OBFMT=YES to a terminal that does not support outboard formatting, BMS ignores the OBFMT operand.

See "Using batch data interchange" on page 383 for more information about programming some of the devices that support outboard formatting.

Part 7. CICS management functions

Chapter 43. Interval control	557
Expiration times	558
Request identifiers	559
Chapter 44. Task control	561
Controlling sequence of access to resources	562
Chapter 45. Program control.	565
Program linking	566
Application program logical levels	566
Link to another program expecting return	566
Passing data to other programs	567
COMMAREA	567
Channels	569
INPUTMSG.	569
Using the INPUTMSG option on the RETURN command	571
Using mixed addressing modes	571
Using LINK to pass data	572
Using RETURN to pass data	574
Chapter 46. Storage control	577
Overview of CICS storage protection and transaction isolation	578
Storage protection	578
Storage categories	579
Transaction isolation	579
Reducing system outages	580
Protecting application data	580
Protecting CICS from being passed invalid addresses	580
Aiding application development	580
Defining the storage key for applications	580
System-wide storage areas	581
Task lifetime storage	581
Program working storage specifically for exit and PLT programs	581
Passing data by a COMMAREA	581
The GETMAIN command	582
Selecting the execution and storage key	583
User-key applications	584
CICS-key applications	584
Tables.	586
Map sets and partition sets	586
Storage protection exception conditions	587
Using transaction isolation	587
MVS subspaces	589
Subspaces and basespaces for transactions	589
The common subspace and shared storage.	590
Chapter 47. Transient data control	593
Intrapartition transient data queues	593
Extrapartition queues	594
Indirect queues	595
Automatic transaction initiation (ATI)	595
Chapter 48. Temporary storage control	597

Temporary storage queues	597
Typical uses of temporary storage control	598

Chapter 43. Interval control

The CICS interval control services provide functions that are related to time.

Java and C++

The application programming interface described in this chapter is the EXEC CICS API, which is not used in Java programs. For information about Java programs using the JCICS classes to access Interval Control services, see the *Java Applications in CICS* component of the CICS Information Center and the JCICS Javadoc html documentation. For information about C++ programs using the CICS C++ classes, see the *CICS C++ OO Class Libraries* manual.

Using interval control commands, you can:

- Start a task at a specified time or after a specified interval, and pass data to it (START command).
- Retrieve data passed on a START command (RETRIEVE command).
- Delay the processing of a task (DELAY command).
- Request notification when a specified time has expired (POST command).
- Wait for an event to occur (WAIT EVENT command).
- Cancel the effect of previous interval control commands (CANCEL command).
- Request the current date and time of day (ASKTIME command).
- Select the format of date and time (FORMATTIME command). Options are available that help you to handle dates in the twenty-first century. Programming information about these is in the *CICS Application Programming Reference* manual .

Note: Do not use EXEC CICS START TRANSID() TERMID(EIBTRMID) to start a remote transaction. Use EXEC CICS RETURN TRANSID() IMMEDIATE instead. START, used in this way, ties up communications resources unnecessarily and can lead to performance degradation across the connected regions.

If you use WAIT EVENT, START, RETRIEVE with the WAIT option, CANCEL, DELAY, or POST commands, you could create inter-transaction affinities that adversely affect your ability to perform dynamic transaction routing.

Storage for the timer-event control area on WAIT EVENT must reside in shared storage if you have specified ISOLATE(YES).

If CICS is executing with or without transaction isolation, CICS checks that the timer-event control area is not in read-only storage.

To help you identify potential problems with programs that issue these commands, you can use the CICS Interdependency Analyzer. See the *CICS Interdependency Analyzer for z/OS User's Guide and Reference* for more information about this utility and Chapter 15, "Affinity," on page 221 for more information about transaction affinity.

This chapter describes:

- "Expiration times" on page 558
- "Request identifiers" on page 559

Expiration times

The time at which a time-controlled function is to be started is known as the **expiration time**. You can specify expiration times absolutely, as a time of day (using the **TIME** option), or as an interval that is to elapse before the function is to be performed (using the **INTERVAL** option). For **DELAY** commands, you can use the **FOR** and **UNTIL** options; and for **POST** and **START** commands, you can use the **AFTER** and **AT** options. See the *CICS Application Programming Reference* manual for programming information about these commands.

Note: The C and C++ languages do not provide the support for the packed decimal types used by the **TIME** and **INTERVAL** options.

You use an **interval** to tell CICS when to start a transaction in a specified number of hours, minutes, and seconds from the current time. A nonzero **INTERVAL** value always indicates a time in the future—the current time plus the interval you specify. The hours may be 0–99, but the minutes and seconds must not be greater than 59. For example, to start a task in 40 hours and 10 minutes, you would code:

```
EXEC CICS START INTERVAL(401000)
```

You can use an **absolute time** to tell CICS to start a transaction at a specific time, again using hhmmss. For example, to start a transaction at 3:30 in the afternoon, you would code:

```
EXEC CICS START TIME(153000)
```

An absolute time is always relative to the midnight before the current time and may therefore be earlier than the current time. **TIME** may be in the future or the past relative to the time at which the command is executed. CICS uses the following rules:

- If you specify a task to start at any time within the previous six hours, it starts immediately. This happens regardless of whether the previous six hours includes a midnight. For example:

```
EXEC CICS START TIME(123000)
```

This command, issued at 05:00 or 07:00 on Monday, expires at 12:30 on the same day.

```
EXEC CICS START TIME(020000)
```

This command, issued at 05:00 or 07:00 on Monday expires immediately because the specified time is within the preceding six hours.

```
EXEC CICS START TIME(003000)
```

This command, issued at 05:00 on Monday, expires immediately because the specified time is within the preceding six hours. However, if it is issued at 07:00 on Monday, it expires at 00:30 on Tuesday, because the specified time is not within the preceding six hours.

```
EXEC CICS START TIME(230000)
```

This command, issued at 02:00 on Monday, expires immediately because the specified time is within the preceding six hours.

- If you specify a time with an hours component that is greater than 23, you are specifying a time on a day following the current one. For example, a time of 250000 means 1 a.m. on the day following the current one, and 490000 means 1 a.m. on the day after that.

If you do not specify an expiration time or interval option on DELAY, POST, or START commands, CICS responds using the default of INTERVAL(0), which means immediately.

Because each end of an intersystem link may be in a different time zone, you should use the INTERVAL form of expiration time when the transaction to be started is in a remote system.

If the system fails, the times associated with unexpired START commands are remembered across the restart.

Notes:

1. On a lightly used system, the interval time specified can be exceeded by as much as a quarter of a second.
2. If your expiration time falls within a possible CICS shutdown, you should consider whether your task should test the status of CICS before attempting to run. You can do this using the CICSSTATUS option of INQUIRE SYSTEM. INQUIRE SYSTEM CICSSTATUS is described in the *CICS System Programming Reference*. During a normal shutdown, your task could run at the same time as the PLT programs with consequences known only to you.

Request identifiers

As a means of identifying the request and any data associated with it, a unique request identifier is assigned by CICS to each DELAY, POST, and START command. You can specify your own request identifier by means of the REQID option. If you do not, CICS assigns (for POST and START commands only) a unique request identifier and places it in field EIBREQID in the EXEC interface block (EIB). You should specify a request identifier if you want the request to be canceled at some later time by a CANCEL command.

Chapter 44. Task control

The CICS task control facility provides functions that synchronize task activity, or that control the use of resources.

Java and C++

The application programming interface described in this chapter is the EXEC CICS API, which is not used in Java programs. For information about Java programs using the JCICS classes to access Task Control services CICS, see *Java Applications in CICS* in the CICS Information Center and the JCICS Javadoc html documentation. For information about C++ programs using the CICS C++ classes, see the *CICS C++ OO Class Libraries* manual.

CICS assigns priorities based on the value set by the CICS system programmer. Control of the processor is given to the highest-priority task that is ready to be processed, and is returned to the operating system when no further work can be done by CICS or by your application programs.

You can:

- Suspend a task (SUSPEND command) to enable tasks of higher priority to proceed. This can prevent processor-intensive tasks from monopolizing the processor. When other eligible tasks have proceeded and terminated or suspended processing, control is returned to the issuing task; that is, the task remains dispatchable.
- Schedule the use of a resource by a task (ENQ and DEQ commands). This is sometimes useful in protecting a resource from concurrent use by more than one task; that is, by making that resource serially reusable. Each task that is to use the resource issues an enqueue command (ENQ). The first task to do so has the use of the resource immediately but, if a HANDLE CONDITION ENQBUSY command has not been issued, subsequent ENQ commands for the resource, issued by other tasks, result in those tasks being suspended until the resource is available.

If the NOSUSPEND option is coded on an ENQ command, control is always returned to the next instruction in the program. By inspecting the contents of the EIBRESP field, you can see whether the ENQ command was successful or not.

Each task using a resource should issue a dequeue command (DEQ) when it has finished with it. However, when using the enqueue/dequeue mechanism, there is no way to guarantee that two or more tasks issuing ENQ and DEQ commands issue these commands in a given sequence relative to each other. For a way to control the sequence of access, see “Controlling sequence of access to resources” on page 562.

- Change the priority assigned to a task (CHANGE TASK PRIORITY command).
- Wait for events that post MVS format ECBs when they complete.

Two commands are available, WAITCICS and WAIT EXTERNAL. These commands cause the issuing task to be suspended until one of the ECBs has been posted; that is, until one of the events has occurred. The task can wait on one or more ECBs. If it waits on more than one, it is dispatchable as soon as one of the ECBs is posted. You must ensure that each ECB is cleared (set to binary zeros) no later than the earliest time it could be posted. CICS cannot do this for you. If you wait on an ECB that has been previously posted and is not

subsequently cleared, your task is not suspended and continues to run as though WAITCICS or WAIT EXTERNAL had not been issued.

WAIT EXTERNAL usually has less overhead, but the associated ECBs must always be posted using the MVS POST facility or by an optimized post (using the compare and swap (CS) instruction). They must never be posted by any other method. If you are in any doubt about the method of posting, use a WAITCICS command. When dealing with ECBs passed on a WAIT EXTERNAL command, CICS extends the ECBs and uses the MVS POST exit facility. A given ECB must not be waited on by more than one task at once (or appear twice in one task's ECBLIST). Failure to follow this rule leads to an INVREQ response.

WAITCICS must be used if ECBs are to be posted by any method other than the MVS POST facility or by an optimized post. For example, if your application posts the ECB by moving a value into it, WAITCICS must be used. (The WAITCICS command can also be used for ECBs that are posted using the MVS POST facility or optimized post.) Whenever CICS goes into an MVS WAIT, it passes a list to MVS of all the ECBs being waited on by tasks that have issued a WAITCICS command. The ECBLIST passed by CICS on the MVS WAIT contains duplicate addresses, and MVS abends CICS.

If you use MVS POST, WAIT EXTERNAL, WAITCICS, ENQ, or DEQ commands, you could create inter-transaction affinities that adversely affect your ability to perform dynamic transaction routing.

To help you identify potential problems with programs that issue this command, you can use the CICS Interdependency Analyzer. See the *CICS Interdependency Analyzer for z/OS User's Guide and Reference* for more information about this utility and Chapter 15, "Affinity," on page 221 for more information about transaction affinity.

Controlling sequence of access to resources

If you want a resource to be accessed by two or more tasks in a specific order, instead of ENQ and DEQ commands, use one or more WAITCICS commands in conjunction with one or more hand-posted ECBs.

To hand-post an ECB, a CICS task sets a 4-byte field to either the cleared state of binary zeros, or the posted state of X'40008000'. The task can use a START command to start another task and pass the address of the ECB. The started task receives the address through a RETRIEVE command.

Either task can set the ECB or wait on it. Use the ECB to control the sequence in which the tasks access resources. Two tasks can share more than one ECB if necessary. You can extend this technique to control as many tasks as you wish.

Note: Only one task can wait on a given ECB at any one time.

The example in Figure 117 on page 563 shows how two tasks can sequentially access a temporary storage queue by using hand-posted ECBs and a WAITCICS command.

The example uses two ECBs, (ECB1 and ECB2), addressed by the pointers illustrated in Figure 118 on page 563.

In theory, these tasks could exchange data through the temporary storage queue for ever. In practice, some code would be included to close the process down in an orderly way.

<p>Task A</p> <p>Delete temporary storage queue</p> <p>Clear ECB1 (set to X'00000000')</p> <p>Clear ECB2</p> <p>EXEC CICS START task B (pass addresses of PTR_ECB1_ADDR_LIST and PTR_ECB2_ADDR_LIST</p> <p>LOOP:</p> <p>EXEC CICS WAITCICS</p> <p>ECBLIST(PTR_ECB1_ADDR_LIST)</p> <p>NUMEVENTS(1)</p> <p>Clear ECB1</p> <p>Read TS queue</p> <p>Process data</p> <p>Delete TS queue</p> <p>Write to TS queue</p> <p>Post ECB2</p> <p>Go to start of loop</p>	<p>Task B</p> <p>EXEC CICS RETRIEVE</p> <p>(addresses passed)</p> <p>LOOP:</p> <p>Write to TS queue</p> <p>Post ECB1 (set to X'40008000)</p> <p>EXEC CICS WAITCICS</p> <p>ECBLIST(PTR_ECB2_ADDR_LIST)</p> <p>NUMEVENTS(1)</p> <p>ClearECB2</p> <p>Read TS queue</p> <p>Process data</p> <p>Delete TS queue</p> <p>Go to start of loop</p>
---	---

Figure 117. Two tasks using WAITCICS to control access to a shared resource

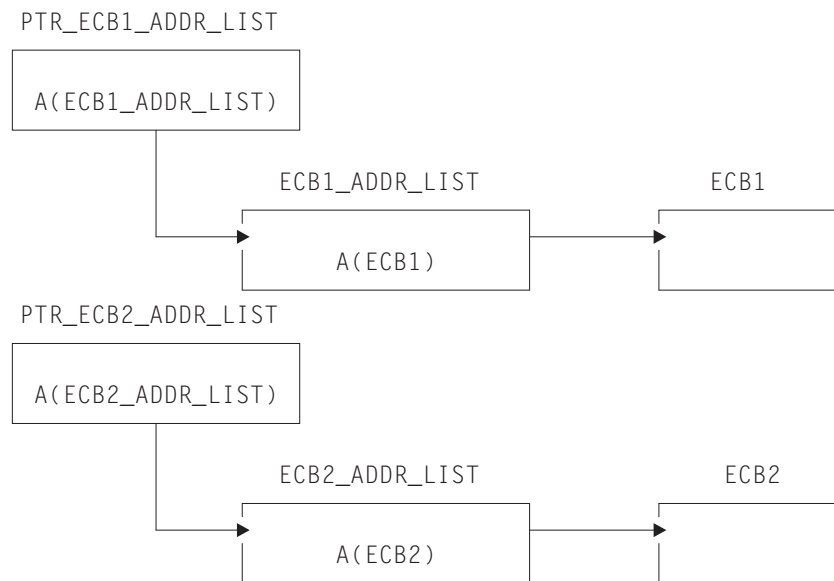


Figure 118. ECB pointers used by WAITCICS example

Chapter 17, “Dealing with exception conditions,” on page 261 describes how the exception conditions that can occur during processing of a task control command are handled.

Chapter 45. Program control

The CICS program control facility governs the flow of control between application programs in a CICS system.

Java and C++

The application programming interface described in this chapter is the EXEC CICS API, which is not used in Java programs. For information about Java programs using the JCICS classes to access program control services, see *Java Applications in CICS* in the CICS Information Center and the JCICS Javadoc html documentation. For information about C++ programs using the CICS C++ classes, see the *CICS C++ OO Class Libraries* manual.

The name of the application referred to in a program control command must have been defined as a program to CICS. You can use program control commands to:

- Link one of your application programs to another, anticipating subsequent return to the requesting program (LINK command). The COMMAREA, CHANNEL, and INPUTMSG options of this command allow data to be passed to the requested application program.
- Link one of your application programs to another program in a separate CICS region, anticipating subsequent return to the requesting program (LINK command). The COMMAREA or CHANNEL option of this command allows data to be passed to the requested application program. This is referred to as distributed program link (DPL). (You cannot use the INPUTMSG and INPUTMSGLEN options of a LINK command when using DPL. See Chapter 30, “CICS intercommunication,” on page 433 for more information about DPL.)
- Transfer control from one of your application programs to another, with no return to the requesting program (XCTL command). The COMMAREA, CHANNEL, and INPUTMSG options of this command allow data to be passed to the requested application program. (You cannot use the INPUTMSG and INPUTMSGLEN options of an XCTL command when using DPL. See Chapter 30, “CICS intercommunication,” on page 433 for more information about DPL.)
- Return control from one of your application programs to another, or to CICS (RETURN command). The COMMAREA, CHANNEL, and INPUTMSG options of this command allow data to be passed to a newly initiated transaction. (You cannot use the INPUTMSG and INPUTMSGLEN options of a RETURN command when using DPL. See Chapter 30, “CICS intercommunication,” on page 433 for more information about DPL.)
- Load a designated application program, table, or map into main storage (LOAD command).

If you use the HOLD option with the LOAD and RELEASE command to load a program, table or map that is not read-only, you could create inter-transaction affinities that could adversely affect your ability to perform dynamic transaction routing.

To help you identify potential problems with programs that issue these commands, you can use the CICS Interdependency Analyzer. See *CICS Interdependency Analyzer for z/OS User's Guide and Reference* for more information about this utility and see Chapter 15, “Affinity,” on page 221 for more information about transaction affinity.

- Delete a previously loaded application program, table, or map from main storage (RELEASE command).

You can use the RESP option to deal with abnormal terminations.

This chapter describes:

- “Program linking”
- “Passing data to other programs” on page 567
- “Using mixed addressing modes” on page 571
- “Using LINK to pass data” on page 572
- “Using RETURN to pass data” on page 574

Program linking

A LINK command is used to pass control from an application program at one logical level to an application program at the next lower logical level.

Application program logical levels

Application programs running under CICS are executed at various logical levels. The first program to receive control within a task is at the highest logical level. When an application program is linked to another, expecting an eventual return of control, the linked-to program is considered to reside at the next lower logical level. When control is simply transferred from one application program to another, without expecting return of control, the two programs are considered to reside at the same logical level.

Link to another program expecting return

If the program receiving control is not already in main storage, it is loaded. When a RETURN command is processed in the linked program, control is returned to the program initiating the link at the next sequential process instruction.

The linked program operates independently of the program that issues the LINK command with regard to handling exception conditions, attention identifiers, and abends. For example, the effects of HANDLE commands in the linking program are not inherited by the linked-to program, but the original HANDLE commands are restored on return to the linking program. You can use the HANDLE ABEND command to deal with abnormal terminations in other link levels. See the *CICS Application Programming Reference* manual for programming information about this command. Figure 119 on page 567 shows the concept of logical levels.

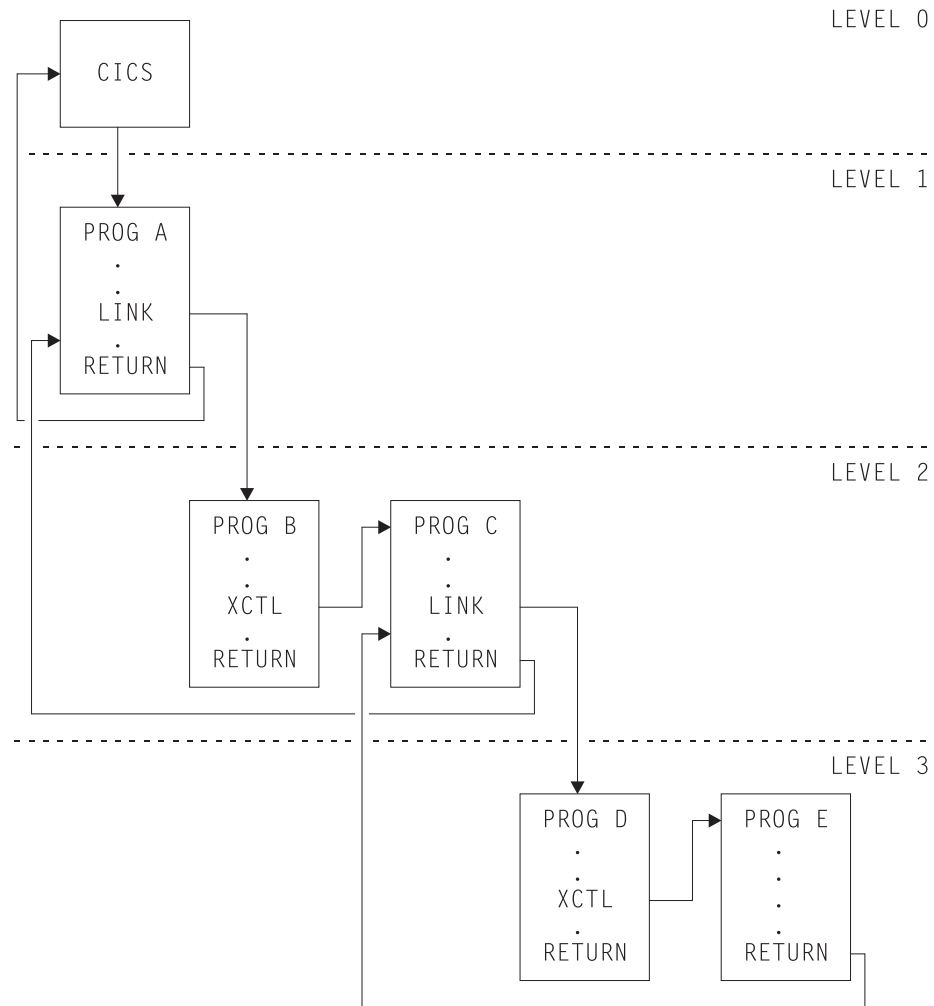


Figure 119. Application program logical levels

Passing data to other programs

You can pass data to another program using the EXEC CICS program control commands, LINK, XCTL, and RETURN and by specifying the COMMAREA, CHANNEL, and INPUTMSG options of those commands. (COMMAREA and CHANNEL are mutually exclusive.)

COMMAREA

The COMMAREA option of LINK and XCTL commands specifies the name of a data area (known as a **communication area**) in which data is passed to the program being invoked.

In a similar manner, the COMMAREA option of a RETURN command specifies the name of a communication area in which data is passed to the transaction identified in the TRANSID option. (The TRANSID option specifies a transaction that is initiated when the next input is received from the terminal associated with the task.)

The invoked program receives the data as a parameter. The program must contain a definition of a data area to allow access to the passed data.

In a receiving COBOL program, you must give the data area the name DFHCOMMAREA. In this COBOL program, if a program passes a COMMAREA as part of a LINK, XCTL, or RETURN command, either the working-storage or the LINKAGE SECTION can contain the data area. A program receiving a COMMAREA should specify the data in the LINKAGE SECTION. This applies when the program is either of the following:

- The receiving program during a LINK or XCTL command where a COMMAREA is passed
- The initial program, where a RETURN command of a previously called task specified a COMMAREA and TRANSID

In a C or C++ program that is receiving a COMMAREA, the COMMAREA must be defined as a pointer to a structure. The program then must issue the ADDRESS COMMAREA command to gain addressability to the passed data.

In a PL/I program, the data area can have any name, but it must be declared as a based variable, based on the parameter passed to the program. The pointer to this based variable should be declared explicitly as a pointer rather than contextually by its appearance in the declaration for the area. This prevents the generation of a PL/I error message. No ALLOCATE statement can be processed within the receiving program for any variable based on this pointer. This pointer must not be updated by the application program.

In an assembler language program, the data area should be a DSECT. The register used to address this DSECT must be loaded from DFHEICAP, which is in the DFHEISTG DSECT.

The receiving data area need not be of the same length as the original communication area; if access is required only to the first part of the data, the new data area can be shorter. However, it must not be longer than the length of the communication area being passed. If it is, your transaction may inadvertently attempt to read data outside the area that has been passed. It may also overwrite data outside the area, which could cause CICS to abend.

To avoid this happening, your program should check whether the length of any communication area that has been passed to it is as expected, by accessing the EIBCALEN field in the EIB of the task. If no communication area has been passed, the value of EIBCALEN is zero; otherwise, EIBCALEN always contains the value specified in the LENGTH option of a LINK, XCTL, or RETURN command, regardless of the size of the data area in the invoked program. You should ensure that the value in EIBCALEN matches the value in the DSECT for your program, and make sure that your transaction is accessing data within that area.

You may also add an identifier to COMMAREA as an additional check on the data that is being passed. This identifier is sent with the sending transaction and is checked for by the receiving transaction.

When a communication area is passed using a LINK command, the invoked program is passed a pointer to the communication area itself. Any changes made to the contents of the data area in the invoked program are available to the invoking program, when control returns to it. To access any such changes, the program names the data area specified in the original COMMAREA option.

When a communication area is passed using an XCTL command, a copy of that area is made unless the area to be passed has the same address and length as

the area that was passed to the program issuing the command. For example, if program A issues a LINK command to program B, which in turn issues an XCTL command to program C, and if B passes to C the same communication area that A passed to B, program C will be passed addressability to the communication area that belongs to A (not a copy of it), and any changes made by C will be available to A when control returns to it.

When a lower-level program, which has been accessed by a LINK command, issues a RETURN command, control passes back one logical level higher than the program returning control. If the task is associated with a terminal, the TRANSID option can be used at the lower level to specify the transaction identifier for the next transaction to be associated with that terminal. The transaction identifier comes into play only after the highest logical level has relinquished control to CICS using a RETURN command and input is received from the terminal. Any input entered from the terminal, apart from an attention key, is interpreted wholly as data. You may use the TRANSID option without COMMAREA when returning from any link level, but it can be overridden on a later RETURN command. If a RETURN command fails at the top level because of an invalid COMMAREA, the TRANSID becomes null. Also, you can specify COMMAREA or IMMEDIATE only at the highest level, otherwise you get an INVREQ with RESP2=2.

In addition, the COMMAREA option can be used to pass data to the new task that is to be started.

The invoked program can determine which type of command invoked it by accessing field EIBFN in the EIB. This field must be tested before any CICS commands are issued. If the program was invoked by a LINK or XCTL command, the appropriate code is found in the EIBFN field. If it was invoked by a RETURN command, no CICS commands have been issued in the task, and the field contains zeros.

Channels

Instead of using a communication area (COMMAREA), a more modern method of transferring data between CICS programs is to use a **channel**. Channels have several advantages over COMMAREAs—see Benefits of channels. Channels can be passed, instead of COMMAREAs, on LINK, XCTL, and RETURN commands.

Channels are described in Enhanced inter-program data transfer: channels as modern-day COMMAREAs.

INPUTMSG

The INPUTMSG option of LINK, XCTL, and RETURN commands is another way of specifying the name of a data area to be passed to the program being invoked. In this case, the invoked program gets the data by processing a RECEIVE command. This option enables you to invoke (“front-end”) application programs that were written to be invoked directly from a terminal, and which contain RECEIVE commands, to obtain initial terminal input.

If program that has been accessed by means of a LINK command issues a RECEIVE command to obtain initial input from a terminal, but the initial RECEIVE request has already been issued by a higher-level program, there is no data for the program to receive. In this case, the application waits on input from the terminal. You can ensure that the original terminal input continues to be available to a linked program by invoking it with the INPUTMSG option.

When an application program invokes another program, specifying INPUTMSG on LINK (or XCTL or RETURN) command, the data specified on the INPUTMSG continues to be available even if the linked program itself does not issue an RECEIVE command, but instead invokes yet another application program. See Figure 120 for an illustration of INPUTMSG.

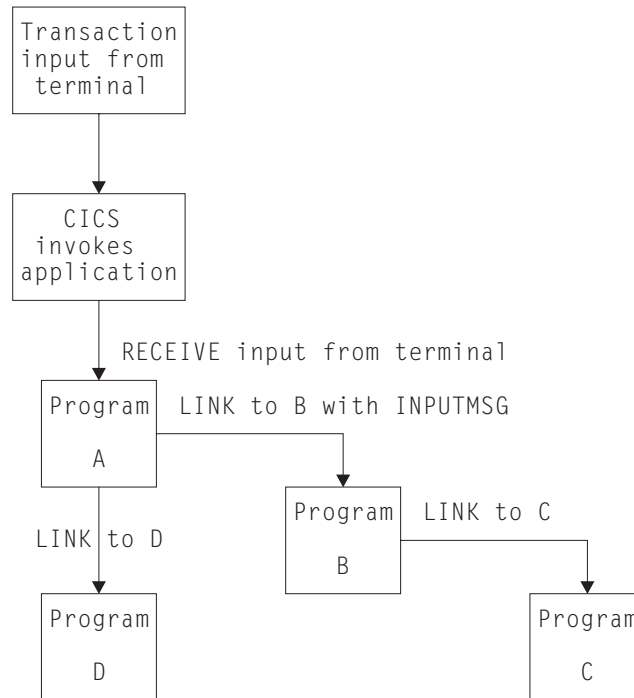


Figure 120. Use of INPUTMSG in a linked chain

Notes:

1. In this example, the “real” first RECEIVE command is issued by program A. By linking to program B with the INPUTMSG option, it ensures that the next program to issue a RECEIVE request can also receive the terminal input. This can be either program B or program C.
2. If program A simply wants to pass on the unmodified terminal input that it received, it can name the same data area for the INPUTMSG option that it used for the RECEIVE command. For example:

```

EXEC CICS RECEIVE
      INTO(TERMINAL-INPUT)
      :
EXEC CICS LINK
      PROGRAM(PROGRAMB)
      INPUTMSG(TERMINAL-INPUT)
      :

```

3. As soon as one program in a LINK chain issues a RECEIVE command, the INPUTMSG data ceases to be available to any subsequent RECEIVE command. In other words, in the example shown, if B issues a RECEIVE request before linking to C, the INPUTMSG data area is not available for C.
4. This method of communicating data from one program to another can be used for any kind of data—it does not have to originate from a user terminal. In our example, program A could move any data into the named data area, and invoke program B with INPUTMSG referencing the data.

5. The “terminal-data” passed on INPUTMSG also ceases to be available when control is eventually returned to the program that issued the link with INPUTMSG. In our example, if C returns to B, and B returns to A, and neither B nor C issues a RECEIVE command, the data is assumed by A to have been received. If A then invokes another program (for example, D), the original INPUTMSG data is no longer available to D, unless the INPUTMSG option is specified.
6. The INPUTMSG data ceases to be available when a SEND or CONVERSE command is issued.

Using the INPUTMSG option on the RETURN command

You can specify INPUTMSG to pass data to the next transaction specified on a RETURN command with the TRANSID option. To do this, RETURN must be issued at the highest logical level to return control to CICS, and the command must also specify the IMMEDIATE option. If you specify INPUTMSG with TRANSID, and do not also specify IMMEDIATE, the next real input from the terminal overrides the INPUTMSG data, which is therefore lost. See the *CICS Application Programming Reference* manual for programming information about the RETURN command.

If you specify INPUTMSG with TRANSID some time after a SEND command, the SEND message is immediately flushed out to the terminal.

The other use for INPUTMSG, on a RETURN command without the TRANSID option, is intended for use with a dynamic transaction routing program. See the *CICS Customization Guide* for programming information about the user-replaceable dynamic transaction routing program.

Using mixed addressing modes

CICS supports the use of LINK, XCTL, and RETURN commands between programs with different addressing modes and between programs with the same addressing mode.

The following restrictions apply to programs passing data using a communication area named by the COMMAREA option:

- Addresses passed within a communication area to an AMODE(31) program must be 31 bits long. Do not use 3-byte addresses with flag data packed into the top byte, unless the called program is specifically designed to ignore the top byte.
- Addresses passed as data to an AMODE(24) program must be below the 16MB line if they are to be interpreted correctly by the called program.

These restrictions apply to the address of the communication area itself, and also to addresses within it. However, a communication area above the 16MB line can be passed to an AMODE(24) subprogram. CICS copies the communication area into an area below the 16MB line for processing. It copies it back again when control returns to the linking program. See Chapter 46, “Storage control,” on page 577 for information about copying CICS-key storage.

CICS does not validate any data addresses passed within a communication area between programs with different addressing modes.

Using LINK to pass data

Figures 121 to 124 show how, in COBOL, C, C++, PL/I, and assembler language, aLINK command causes data to be passed to the program being linked to; an XCTL command is coded in a similar way.

These examples show data being passed in a COMMAREA. For an example of a LINK command that uses a channel to pass data, see Enhanced inter-program data transfer: channels as modern-day COMMAREAs.

```

                                Invoking program
IDENTIFICATION DIVISION.
PROGRAM ID. 'PROG1'.
.
WORKING-STORAGE SECTION.
01 COM-REGION.
   02 FIELD PICTURE X(3).
.
PROCEDURE DIVISION.
   MOVE 'ABC' TO FIELD.
   EXEC CICS LINK PROGRAM('PROG2')
       COMMAREA(COM-REGION)
       LENGTH(3) END-EXEC.
.
                                Invoked program
IDENTIFICATION DIVISION.
PROGRAM-ID. 'PROG2'.
.
LINKAGE SECTION.
01 DFHCOMMAREA.
   02 FIELD PICTURE X(3).
.
PROCEDURE DIVISION.
   IF EIBCALEN GREATER ZERO
   THEN
       IF FIELD EQUALS 'ABC' ...
```

Figure 121. COBOL example—LINK command

```

                                Invoking program
main()
{
    unsigned char field[3];
    memcpy(field, "ABC", 3);
    EXEC CICS LINK PROGRAM("PROG2")
        COMMAREA(field)
        LENGTH(sizeof(field));
}
                                Invoked program
main()
{
    unsigned char *commarea;
    EXEC CICS ADDRESS COMMAREA(commarea) EIB(dfheiptr);
    if (dfheiptr->eibcalen > 0)
    {
        if (memcmp(commarea, "ABC", 3) == 0)
        {
            .
        }
    }
}
```

Figure 122. C example—LINK command. In this example, the COMMAREA contains a character string. For an example of a COMMAREA that contains a structure, see Figure 126 on page 575.

```

                                Invoking program
PROG1: PROC OPTIONS(MAIN);
DCL 1 COM_REGION AUTOMATIC,
      2 FIELD CHAR(3),
.
FIELD='ABC';
EXEC CICS LINK PROGRAM('PROG2')
      COMMAREA(COM_REGION) LENGTH(3);
END;

                                Invoked program
PROG2:
PROC(COMM_REG_PTR) OPTIONS(MAIN);
DCL COMM_REG_PTR PTR;
DCL 1 COM_REGION BASED(COMM_REG_PTR),
      2 FIELD CHAR(3),
.
IF EIBCALEN>0 THEN DO;
    IF FIELD='ABC' THEN ...
.
    END;
END;

```

Figure 123. PL/I example—LINK command

```

                                Invoking program
DFHEISTG DSECT
COMREG   DS 0CL20
FIELD    DS CL3
.
PROG1    CSECT
.
        MVC FIELD,=C'XYZ'
        EXEC CICS LINK
            PROGRAM('PROG2')
            COMMAREA(COMREG) LENGTH(3)
.
        END

                                Invoked program
COMREG   DSECT
FIELD    DS CL3
.
PROG2    CSECT
.
        L  COMPTR,DFHEICAP
        USING COMREG,COMPTR
        CLC FIELD,=C'ABC'
.
        END

```

Figure 124. ASM example—LINK command

Using RETURN to pass data

Figures 125 to 128 show how in COBOL, C, C++, PL/I, and assembler language, a RETURN command is used to pass data to a new transaction.

These examples show data being returned in a COMMAREA. For an example of a RETURN command that uses a channel to return data, see Enhanced inter-program data transfer: channels as modern-day COMMAREAs.

```

                                Invoking program
IDENTIFICATION DIVISION.
PROGRAM-ID. 'PROG1'.

.
WORKING-STORAGE SECTION.
01  TERMINAL-STORAGE.
    02  FIELD PICTURE X(3).
    02  DATAFLD PICTURE X(17).

.
PROCEDURE DIVISION.
    MOVE 'XYZ' TO FIELD.
    EXEC CICS RETURN TRANSID('TRN2')
        COMMAREA(TERMINAL-STORAGE)
        LENGTH(20) END-EXEC.

.
                                Invoked program
IDENTIFICATION DIVISION.
PROGRAM-ID. 'PROG2'

.
LINKAGE SECTION.
01  DFHCOMMAREA.
    02  FIELD PICTURE X(3).
    02  DATAFLD PICTURE X(17).

.
PROCEDURE DIVISION.
    IF EIBCALEN GREATER ZERO
    THEN
        IF FIELD EQUALS 'XYZ'
        MOVE 'ABC' TO FIELD.
    EXEC CICS RETURN END-EXEC.
```

Figure 125. COBOL example—RETURN command

```

                                Invoking program
struct ter_struct
{
    unsigned char field[3];
    unsigned char datafld[17];
};
main()
{
    struct ter_struct ter_stor;
    memcpy(ter_stor.field,"XYZ",3);
    EXEC CICS RETURN TRANSID("TRN2")
        COMMAREA(&ter_stor)
        LENGTH(sizeof(ter_stor));
}

                                Invoked program
struct term_struct
{
    unsigned char field[3];
    unsigned char datafld[17];
};
main()
{
    struct term_struct *commarea;
    EXEC CICS ADDRESS COMMAREA(commarea) EIB(dfheiptr);
    if (dfheiptr->eibcalen > 0)
    {
        if (memcmp(commarea->field, "XYZ", 3) == 0)
            memcpy(commarea->field, "ABC", 3);
    }
    EXEC CICS RETURN;
}

```

Figure 126. C example—RETURN command

```

                                Invoking program
PROG1: PROC OPTIONS(MAIN);
DCL 1 TERM_STORAGE,
    2 FIELD CHAR(3),
.
FIELD='XYZ';
EXEC CICS RETURN TRANSID('TRN2')
    COMMAREA(TERM_STORAGE);
END;

                                Invoked program
PROG2:
PROC(TERM_STG_PTR) OPTIONS(MAIN);
DCL TERM_STG_PTR PTR;
DCL 1 TERM_STORAGE
    BASED(TERM_STG_PTR),
    2 FIELD CHAR(3),
.
IF EIBCALEN>0 THEN DO;
    IF FIELD='XYZ' THEN FIELD='ABC';
END;
EXEC CICS RETURN;
END;

```

Figure 127. PL/I example—RETURN command

Invoking program

```
DFHEISTG DSECT
TERMSTG  DS 0CL20
FIELD    DS CL3
DATAFLD  DS CL17
:
PROG1    CSECT
:
:
        MVC FIELD,=C'ABC'
        EXEC CICS RETURN
        TRANSID('TRN2')
        COMMAREA(TERMSTG)
:
        END
```

Invoked program

```
TERMSTG  DSECT
FIELD    DS CL3
DATAFLD  DS CL17
:
PROG2    CSECT
:
:
        CLC EIBCALEN,=H'0'
        BNH LABEL2
        L  COMPTR,DFHEICAP
        USING TERMSTG,COMPTR
        CLC FIELD,=C'XYZ'
        BNE LABEL1
        MVC FIELD,=C'ABC'
LABEL1    DS 0H
:
:
LABEL2    DS 0H
:
:
        END
```

Figure 128. ASM example—RETURN command

Chapter 46. Storage control

The CICS storage control facility controls requests for main storage to provide intermediate work areas and other main storage needed to process a transaction.

Java and C++

The application programming interface described in this chapter is the EXEC CICS API, which is not used in Java programs. For information about Java programs using the JCICS classes to access storage control services, see *Java Applications in CICS* in the CICS Information Center and the JCICS Javadoc html documentation. For information about C++ programs using the CICS C++ classes, see the *CICS C++ OO Class Libraries* manual.

CICS makes working storage available within each command-level program automatically, without any specific request from the application program, and provides other facilities for intermediate storage, both within and among tasks. Chapter 12, “Design for performance,” on page 165 describes storage within individual programs. If you need working storage in addition to the working storage provided automatically by CICS, however, you can use the following commands:

- GETMAIN to get and initialize main storage
- FREEMAIN to release main storage

You can initialize the acquired main storage to any bit configuration by supplying the INITIMG option on the GETMAIN command; for example, zeros or EBCDIC blanks.

CICS releases all main storage associated with a task when the task is ended normally or abnormally. This includes any storage acquired, and not subsequently released, by your application program, except for areas obtained with the SHARED option. This option of the GETMAIN command prevents storage being released automatically when a task completes.

If you use the GETMAIN command with the SHARED option, and the FREEMAIN command, you could create inter-transaction affinities that adversely affect the ability to perform dynamic transaction routing.

To help you identify potential problems with programs that issue these commands, you can use the CICS Interdependency Analyzer. See the *CICS Interdependency Analyzer for z/OS User's Guide and Reference* for information about this utility and see Chapter 15, “Affinity,” on page 221 for information about transaction affinity.

If there is no storage available when you issue your request, CICS suspends your task until space is available, unless you specify the NOSUSPEND option. While the task is suspended, it may be canceled (timed out) if the transaction definition specifies SPURGE(YES) and DTIMOUT(mmss). NOSUSPEND returns control to your program if storage is not available, allowing you to do alternative processing, as appropriate.

This chapter describes:

- “Overview of CICS storage protection and transaction isolation” on page 578
- “Defining the storage key for applications” on page 580
- “Selecting the execution and storage key” on page 583
- “Using transaction isolation” on page 587
- “MVS subspaces” on page 589

Overview of CICS storage protection and transaction isolation

Storage control is affected by Storage protection introduced in CICS/ESA 3.3 and Transaction isolation introduced in CICS/ESA 4.1.

Storage protection protects CICS code and control blocks from applications, and transaction isolation protects tasks from each other.

The ESA/390 subsystem storage protection facility works in a way that enables you to prevent CICS code and control blocks from being overwritten accidentally by your application programs. It does **not** provide protection against deliberate overwriting of CICS code or control blocks. CICS cannot prevent an application obtaining the necessary access (execution key) to modify CICS storage.

Transaction isolation extends this storage protection to provide protection for transaction data. Accidental overwrites of the transaction data by an application program of another transaction can affect the reliability and availability of your CICS system and the integrity of the data in the system.

The use of storage protection is optional. You choose whether you want to use storage protection facilities by means of CICS system initialization parameters described in the *CICS System Definition Guide*. For information about transaction isolation, see “Transaction isolation” on page 579.

Storage protection

CICS allows you to run your application programs in either user-key or CICS-key storage. (See “Storage categories” on page 579 for definitions of the terms user key and CICS key.) CICS storage is automatically protected from being overwritten by application programs that execute in user-key storage (the default). The concept of isolating CICS code and control blocks (CICS internal data areas) from user application programs is illustrated in Figure 129.

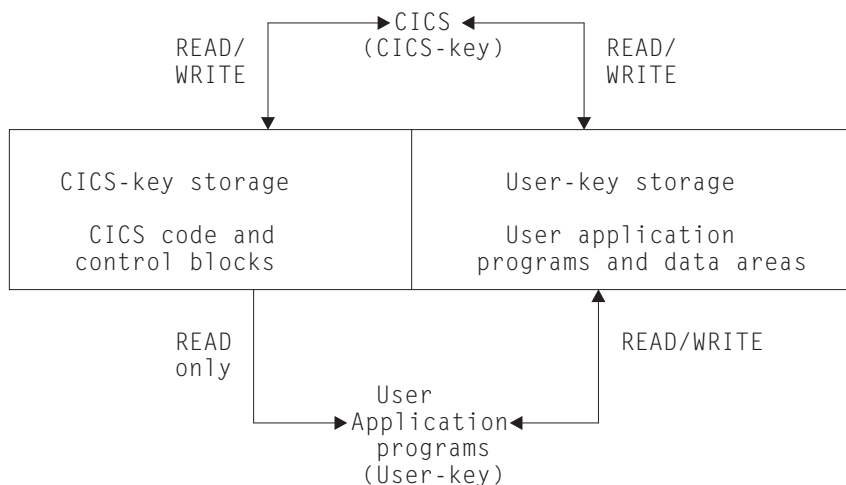


Figure 129. Protecting CICS code and control blocks from user application programs

The terms in Figure 129 relating to storage keys and execution keys are explained under “Storage categories” on page 579.

Storage categories

When you are running with the storage protection facility active, CICS separates storage into two categories:

CICS-key storage

is used for CICS system code and control blocks and, at the discretion of the installation, other code and data areas that require protection from overwriting.

In a CICS region with transaction isolation active, a CICS-key program has read/write access to CICS-key and user-key storage.

User-key storage

is where application programs and their data areas normally reside.

There are two associated execution modes:

1. CICS system programs run in **CICS key**. CICS-key execution allows a program read-write access to both CICS-key and user-key storage.
2. Application programs normally execute in **user key**. User-key execution allows a program read-write access to user-key storage, but only read access to CICS-key storage.

The terms “user key” and “CICS key” thus apply both to storage and to the execution of programs with respect to that storage. They are reflected in the resource definition keywords used in TRANSACTION definitions. See the *CICS Resource Definition Guide* for more information.

The execution key controls the type of access your application programs have to CICS-key storage. The default is that application programs are given control in user key. You should define CICS key only for those programs where it is essential that they execute in CICS key. The programs you might select to run in CICS key are typically those that are written by system programmers, and are usually designed to provide special function in support of user applications. Such programs are generally considered to be an extension of CICS rather than part of an application. Some examples of such programs are described in “CICS-key applications” on page 584.

The storage protection facility does not protect CICS code and control blocks from being overwritten by this type of program, or by ordinary application programs that you choose to execute in CICS key.

Defining the execution key: To run your programs in CICS key, you should use the execution key parameter (EXECKEY) on the program resource definition. See “Selecting the execution and storage key” on page 583 for an explanation of EXECKEY. The EXECKEY parameter determines the key in which CICS passes control to an application program.

Transaction isolation

Transaction isolation uses the MVS subspace group facility to offer protection between transactions. This ensures that an application program associated with one transaction cannot accidentally overwrite the data of another transaction.

Some of the benefits of transaction isolation, and its associated support are:

- Reducing system outages
- Protecting application data
- Protecting CICS from application programs that pass invalid addresses
- Aiding application development

Reducing system outages

Transaction isolation prevents data corruption and unplanned CICS system outages caused by coding errors in user-key application programs that cause the storage of user-key transactions to be accidentally overwritten. Prevention of accidental transaction data overwrites significantly improves the reliability and availability of CICS regions.

Protecting application data

If an application program overwrites CICS code or data, CICS can fail as a result. If an application program overwrites another application program's **code**, that other application program is likely to fail. Whereas this is a serious interruption in a production region, the effect is immediate and the program can generally be recovered so that the terminal user can retry the failed transaction. However, if an application program of one transaction overwrites the **data** of another transaction, the results often are not immediately apparent; the erroneous data can be written to a database and the error may remain undetected until later, when it may be impossible to determine the cause of the error. The consequences of a data overwrite are often much more serious than a code overwrite.

Protecting CICS from being passed invalid addresses

CICS also protects itself against applications that pass invalid addresses that would result in CICS causing storage violations. This occurs when an application program issues an EXEC CICS command that causes CICS to modify storage on the program's behalf, but the program does not own the storage. In earlier releases, CICS did not check ownership of the storage referenced by the passed address, and executed such commands with consequent storage violations.

CICS validates the start address of the storage, and establishes that the application program has write access to the storage that begins with that address, before executing the command.

This address checking is controlled using the CMDPROT system initialization parameter. If a program passes an invalid address to CICS as an output field on the API, an AEYD abend occurs. It is completely independent of storage protection and transaction isolation.

Aiding application development

Transaction isolation aids application development in the testing and debugging phase. If an application tries to overwrite CICS or another application, or if it tries to pass a storage address it does not own for CICS to write to, CICS immediately abends the task and reports the rogue program's name and the area it tried to overwrite. This saves much time trying to debug what is a common problem in application development environments.

Defining the storage key for applications

CICS enables you to choose between user-key storage and CICS-key storage for a number of CICS data areas and application program data areas that your applications can use. Depending on the data area, you select the storage key by:

- System initialization parameters
- Resource definition option
- Selecting an option on the GETMAIN command

Defining the storage key for storage areas that your applications need to access is described in the following sections.

System-wide storage areas

For each CICS region, your installation can choose between user-key and CICS-key storage for the common work area (CWA) and for the terminal control table user areas (TCTUAs). If these areas are in user-key storage, all programs have read-write access to them; if they are in CICS-key storage, user-key application programs are restricted to read-only access. The storage keys for the CWA and the TCTUAs are set by the system initialization parameters CWAKEY and TCTUAKEY, respectively. In both cases the default option is that CICS obtains user-key storage.

See the *CICS System Definition Guide* for information about how to specify these and other storage-protection-related system initialization parameters.

Task lifetime storage

You can also specify whether user-key or CICS-key storage is used for the storage that CICS acquires at transaction attach time, and for those elements of storage directly related to the individual application programs in a transaction. You do this by means of the TASKDATAKEY option on the transaction resource definition. This governs the type of storage allocated for the following storage areas:

- The transaction work area (TWA) and the EXEC interface block (EIB)
- The copies of working storage that CICS obtains for each execution of an application program
- Any storage obtained for an application program in response to:
 - Explicit storage requests by means of an GETMAIN command
 - Implicit storage requests as a result of a CICS command that uses the SET option

For information about how to specify the TASKDATAKEY parameter, see the *CICS Resource Definition Guide*.

Figure 130 on page 582 shows what TASKDATAKEY controls for both task lifetime storage and program working storage.

See the *CICS Application Programming Reference* manual for programming information about EXEC CICS commands; see the *CICS Resource Definition Guide* for information about specifying the TASKDATAKEY option on the transaction resource definition.

Program working storage specifically for exit and PLT programs

CICS uses the TASKDATAKEY option of the calling transaction to determine the storage key for the storage acquired for global user exits, task-related user exits, user-replaceable modules, and PLT programs. For programming information about storage key, including details of how this affects the different types of program, see the *CICS Customization Guide*.

Passing data by a COMMAREA

In a pseudoconversational application, CICS ensures that a COMMAREA you specify on a RETURN command is always accessible in read-write mode to the next program in the conversation. The same is true when passing a COMMAREA within a transaction that comprises more than one program (using a LINK or XCTL command). CICS ensures that the target program has read-write access to the COMMAREA.

The GETMAIN command

The GETMAIN command provides USERDATAKEY and CICSDATAKEY options to enable the application program to explicitly request user-key or CICS-key storage, regardless of the TASKDATAKEY option specified on the associated transaction resource definition. For example, this option allows application programs, which are executing with TASKDATAKEY(CICS) specified, to obtain user-key storage for passing to, or returning to, a program executing in user key.

CICS-key storage obtained by GETMAIN commands issued in a program defined with EXECKEY(CICS) can be freed explicitly only if the FREEMAIN command is issued by a program defined with EXECKEY(CICS). If an application program defined with EXECKEY(USER) attempts to free CICS-key storage using FREEMAIN commands, CICS returns the INVREQ condition. However, an application can free user-key storage with FREEMAIN commands regardless of the EXECKEY option.

All task lifetime storage acquired by an application, whether in CICS key or user key, is freed by CICS at task termination. You can also specify STORAGECLEAR(YES) on this option of the associated transaction resource definition. This clears the storage and so prevents another task accidentally viewing sensitive data.

For programming information about commands, see the *CICS Application Programming Reference* manual ; for information about defining resources, see the *CICS Resource Definition Guide*.

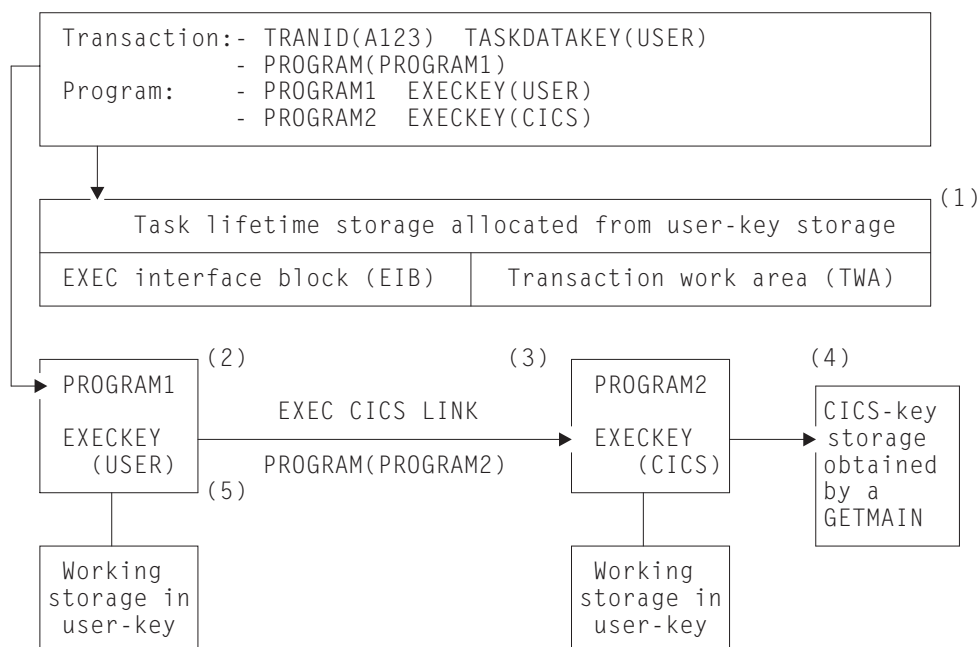


Figure 130. Illustration of the use of the TASKDATAKEY and EXECKEY options

Notes:

1. The TASKDATAKEY option ensures the TWA and EIB are allocated from user-key storage, required for PROGRAM1, which executes in user key—specified by EXECKEY(USER).
2. PROGRAM1 executes in user key (controlled by EXECKEY), and has its working storage obtained in user-key storage (controlled by the TASKDATAKEY

option). Any other storage the program obtains by means of GETMAIN commands or by using the SET option on a CICS command is also obtained in user-key storage.

3. PROGRAM2 executes in CICS key (controlled by EXECKEY), but has its working storage obtained in user-key storage, which again is controlled by the TASKDATAKEY option.
4. PROGRAM2 issues an explicit GETMAIN command using the CICS DATAKEY option and, because it executes in CICS key, can store data into the CICS-key protected storage before returning control to PROGRAM1.
5. PROGRAM1 cannot write to the CICS-key protected storage that PROGRAM2 acquired, but can read what PROGRAM2 wrote there.

When deciding whether you need to specify EXECKEY(CICS) and TASKDATAKEY(CICS), you must consider all the reasons that make these options necessary.

Programs that modify their storage protection key should ensure they are running in the correct key when attempting to access storage. CICS can only use the EXECKEY defined in the program definition when invoking a program.

Selecting the execution and storage key

When you are running CICS with storage protection, the majority of your application programs should execute in user key, with all their storage obtained in user key. You only need to define EXECKEY(CICS) on program definitions, and TASKDATAKEY(CICS) on the associated transaction definitions, for those programs that use facilities that are not permitted in user key, or for any special “system-type” transactions or vendor packages.

You should only specify TASKDATAKEY(CICS) for those transactions where all the component programs have EXECKEY(CICS), and for which you want to protect their task lifetime and working storage from being overwritten by user-key applications. For example, the CICS-supplied transactions such as CEDF are defined with TASKDATAKEY(CICS).

Note that you cannot specify EXECKEY(USER) on any programs that form part of a transaction defined with TASKDATAKEY(CICS) because, in this situation, a user-key program would not be able to write to its own working storage. Transactions abend with an AEZD abend if any program is defined with EXECKEY(USER) within a transaction defined with TASKDATAKEY(CICS), regardless of whether storage protection is active.

You cannot define a program so that it inherits its caller’s execution key. The execution key and data storage keys are derived for each program from its program and associated transaction resource definitions respectively, which you either specify explicitly or allow to default; the default is always user key. Table 45 on page 584 summarizes the various combinations of options.

Table 45. Combinations of KEY options

EXECKEY	TASKDATAKEY	Recommended usage and comments
USER	USER	For normal applications using the CICS API
USER	CICS	Not permitted. CICS abends any program defined with EXECKEY(USER) invoked under a transaction defined with TASKDATAKEY(CICS).
CICS	USER	For programs that need to issue restricted MVS requests or modify CICS-key storage.
CICS	CICS	For transactions (and component programs) that function as extensions to CICS, such as the CICS-supplied transactions, or which require the same protection.

User-key applications

For most applications you should define your programs with EXECKEY(USER), and the related transactions with TASKDATAKEY(USER). To obtain the maximum benefits from the CICS storage protection facility, you are recommended to run your application programs in user key storage. Specifying USER on these options has the following effect:

EXECKEY(USER)

This specifies that CICS is to give control to the program in user key when it is invoked. Programs defined with EXECKEY(USER) are restricted to read-only access to CICS-key storage. These include:

- Storage belonging to CICS itself
- CICS-key storage belonging to user transactions defined with TASKDATAKEY(CICS)
- Application programs defined with EXECKEY(CICS) and thus loaded into CICS-key storage
- In a CICS region where transaction isolation is active, a user-key program has read/write access to the user-key task-lifetime storage of its own transaction and any shared DSA storage

TASKDATAKEY(USER)

This specifies that all task lifetime storage, such as the transaction work area (TWA) and the EXEC interface block (EIB), is obtained from the user-key storage.

It also means that all storage directly related to the programs within the transaction is obtained from user-key storage.

However, user-key programs of transactions defined with ISOLATE(YES) have access only to the user-key task-lifetime storage of their own task.

USER is the default for both the EXECKEY and TASKDATAKEY options, therefore you do not need to make any changes to resource definitions for existing application programs.

CICS-key applications

Most application programs can be defined with EXECKEY(USER), which is the default value, and this is the option you are recommended to use in the majority of

cases. These include programs that use DL/I or DB2 and programs that access vendor products through the resource manager interface (RMI) or a LINK command.

However, some application programs need to be defined with EXECKEY(CICS) because they need to use certain facilities that are listed later. Widespread use of EXECKEY(CICS) diminishes the protection offered by the storage protection facility because there is no protection of CICS code and control blocks from being overwritten by application programs that execute in CICS key. The ISOLATE attribute in the transaction definition does not provide any protection against application programs that execute in CICS key—that is, from programs defined with EXECKEY(CICS). Any application program causing a protection exception when defined with EXECKEY(USER) must be examined to determine why it is attempting to modify storage it is not allowed to modify. You should change a program's definition to EXECKEY(CICS) only if you are satisfied that the application program legitimately uses the facilities described below.

- The program uses MVS macros or services directly, rather than through the CICS API. The only MVS macros that are supported in user-key programs are SPIE, ESPIE, POST, WAIT, WTO, and WTOR. It is also possible to issue GTF trace requests from an EXECKEY(USER) program. If a program uses any other MVS macro or service, it must be defined with EXECKEY(CICS). Some particular examples are:
 - Use of dynamic allocation (DYNALLOC macro, SVC 99)
 - Use of MVS GETMAIN and FREEMAIN or STORAGE requests
 - Use of MVS OPEN, CLOSE, or other file access requests

Direct use of some MVS macros and services is undesirable, even in a CICS application defined with EXECKEY(CICS). This is because they may cause MVS to suspend the whole CICS region until the request is satisfied.

Some COBOL, PL/I, C, and C++ language statements, and compiler options, cause operating system functions to be invoked. See Chapter 4, “Programming in COBOL,” on page 21, Chapter 5, “Programming in C and C++,” on page 43, and Chapter 6, “Programming in PL/I,” on page 53 for information about which of these should not be used in CICS application programs. It is possible that some of these functions may have worked in previous releases of CICS, or at least may not have caused the application to fail. They **do not work** when the program is defined with EXECKEY(USER). When the use of prohibited options or statements is the cause of a protection exception, you should remove these from the program rather than simply redefine the program with EXECKEY(CICS). The use of prohibited statements and options can have other side effects on the overall execution of CICS, and these should be removed.

- The program needs to modify the CWA, and the CWA is in CICS-key storage (CWAKEY=CICS).

If you decide to protect the CWA by specifying CWAKEY(CICS), you should restrict the programs that are permitted to modify the CWA to as few as possible, perhaps only one. See “Using the common work area (CWA)” on page 183 for information about how you can control access to a protected CWA.

- The program needs to modify the TCTUA, and the TCTUAs are in CICS-key storage (TCTUAKEY=CICS).

See “Using the TCTTE user area (TCTUA)” on page 186 for information about using TCTUAs in a storage protection environment.

- The program can be invoked from PLT programs, from transactions defined with TASKDATAKEY(CICS), from task-related or global user exits programs, or from user-replaceable programs.

- The program modifies CICS control blocks—for example, some vendor products that do need to manipulate CICS control blocks. These must be defined with EXECKEY(CICS).
- The program provides user extensions to CICS and requires protection and data access similar to CICS system code. For example, you may consider that such programs are a vital part of your CICS installation, and that their associated storage, like CICS storage, should be protected from ordinary application programs.
- CICS always gives control in CICS key to the following types of user-written program, regardless of the option specified on their program resource definitions:
 - Global user exits (GLUEs)
 - Task-related user exits (TRUEs)
 - User-replaceable modules (URMs)
 - Program list table (PLT) programs

CICS ensures that when control is passed to a PLT program, a global or task-related user exit, or a user-replaceable program, the first program so invoked executes in CICS key, regardless of the EXECKEY specified on its program resource definition. However, if this first program LINKs or XCTLs to other programs, these programs execute under the key specified in their program definitions. If these subsequent programs are required to write to CICS-key data areas, as often occurs in this type of situation, they must be defined as EXECKEY(CICS).

In a CICS region with transaction isolation active, these TRUEs and GLUEs run in either base space or subspace (see “MVS subspaces” on page 589), depending on the current mode when CICS gives control to the exit program. They can also modify any application storage. The URM and PLT programs execute in base space.

For programming information about the execution of GLUEs, TRUEs, URM, and PLT programs in a CICS region running with storage protection, see the *CICS Customization Guide*.

If two transactions have an affinity by virtue of sharing task lifetime storage, the transactions must be defined as ISOLATE(NO), or the programs must be defined as EXECKEY(CICS). You can use the CICS Interdependency Analyzer to check the causes of transaction affinity. See the *CICS Interdependency Analyzer for z/OS User's Guide and Reference* for more information about this utility. The first of these options is the recommended option, because CICS system code and data is still protected.

Tables

In addition to executable programs, you can define tables, map sets, and partition sets as program resources. EXECKEY has less relevance to these objects, because they are not actually executed. However, EXECKEY does control where non-executable objects are loaded, and thus affects whether other programs can store into them.

Map sets and partition sets

Map sets are not reentrant (BMS itself updates fields in maps when calculating absolute screen positions). However, map sets should not be modified by application programs; they must be modified only by CICS, which always executes in CICS key. CICS always loads map sets and partition sets into CICS-key storage.

Storage protection exception conditions

If an application program executing in user key attempts to modify CICS-key storage, a protection exception occurs. The protection exception is processed by normal CICS program error handling, and the offending transaction abends with an ASRA abend. The exception condition appears to the transaction just as if it had attempted to reference any other protected storage. CICS error handling checks whether the reference is to a CICS-key dynamic storage area (DSA), and sends a message (DFHSR0622) to the console. Otherwise, CICS does not treat the failure any differently from any other ASRA abend. See *CICS Problem Determination Guide* for more information about the storage protection exception conditions.

Using transaction isolation

Transaction isolation is built on top of storage protection, which means that STGPROT=YES must be specified. Transaction isolation makes use of parameters introduced by storage protection, these being EXECKEY and TASKDATAKEY.

In addition to being able to specify the storage and execution key for user transactions, you can also specify whether you want transaction isolation. You can control transaction isolation globally for the whole CICS region by means of the TRANISO system initialization parameter. For individual transactions, the ISOLATE option of the transaction resource definition allows you to specify the level of protection that should apply to each transaction and program.

ISOLATE [YES or NO]

The defaults for these options mean that, in most cases, no changes to resource definition are needed for existing applications. However, where necessary, protection can be tailored to allow transactions to continue to function where they fail to meet the criteria for full protection, which is the default. This means that the transaction's user-key task lifetime storage is protected from the user-key programs of other transactions, but not from CICS-key programs. See Figure 131 on page 588 for an illustration of this.

A user-key program invoked by transaction A (TXNA) may read and write to TXNA's user-key task lifetime storage and to shared user storage. Moreover, TXNA has no access to transaction B's (TXNB) user-key task lifetime storage.

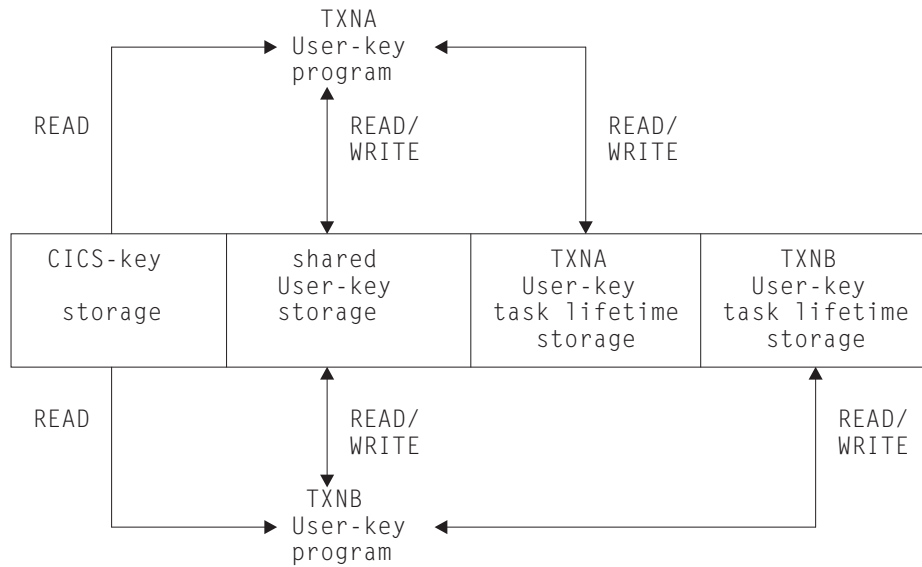


Figure 131. Two transactions defined as ISOLATE(YES)

If a transaction is defined as ISOLATE(NO), its user-key task lifetime is visible to all other transactions also defined as ISOLATE(NO). It is, however, protected from transactions defined as ISOLATE(YES).

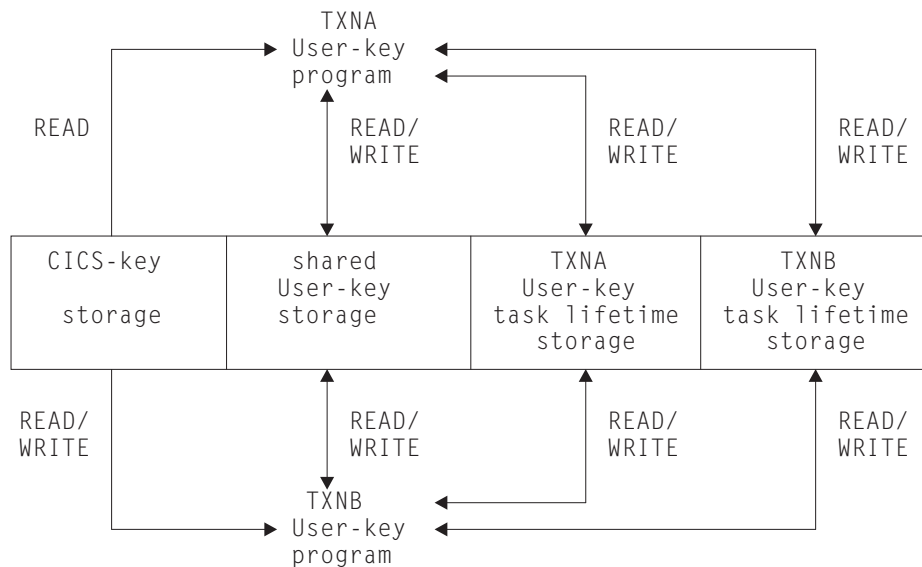


Figure 132. Two transactions defined as ISOLATE(NO) with read/write access to each other's task lifetime storage

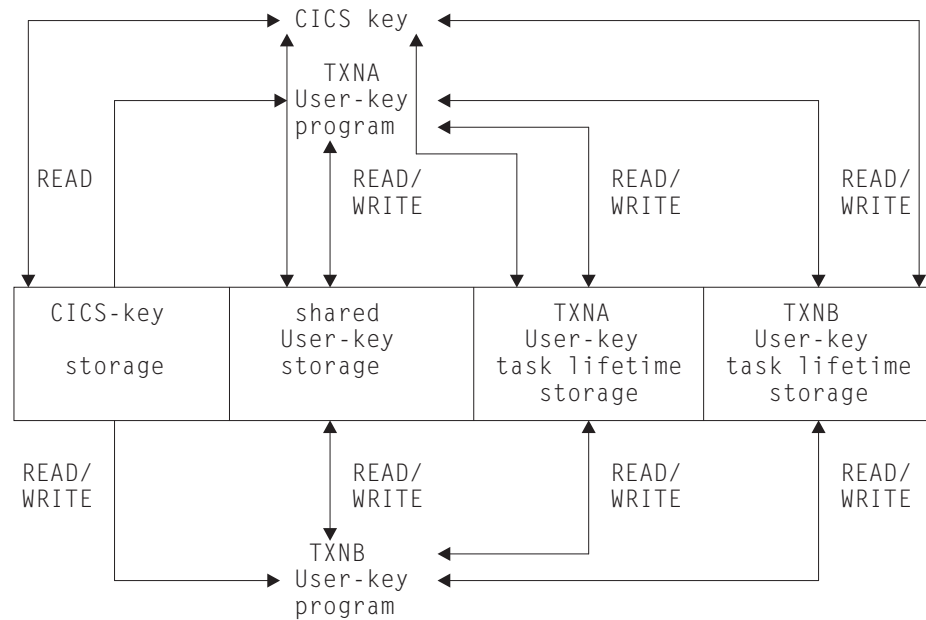


Figure 133. Two transactions defined as ISOLATE(YES) to a CICS-key program that has read/write access to both CICS- and user-key storage

MVS subspaces

MVS/ESA 5.2 introduces the subspace group facility, which can be used for storage isolation to preserve data integrity within an address space.

The subspace-group facility uses hardware to provide protection for transaction data. A subspace-group is a group of subspaces and a single base space, where the base space is the normal MVS address space as in releases prior to MVS/ESA 5.1.

The subspace-group facility provides a partial mapping of the underlying base space, so that only specified areas of storage in the base space are exposed in a particular subspace. Thus each subspace represents a different subset of the storage in the base space. Transaction isolation, when specified, ensures that programs defined with EXECKEY(USER) execute in their own subspace, with appropriate access to any shared storage, or to CICS storage. Thus a user transaction is limited to its own “view” of the address space.

Programs defined with EXECKEY(CICS) execute in the base space, and have the same privileges as in CICS/ESA 3.3.

Subspaces and basespaces for transactions

In general, transaction isolation ensures that user-key programs are allocated to separate (unique) subspaces, and have:

- Read and write access to the user-key task-lifetime storage of their own tasks, which is allocated from one of the user dynamic storage areas (UDSA or EUDSA)
- Read and write access to shared storage; that is, storage obtained by GETMAIN commands with the SHARED option (SDSA or ESDSA)

- Read access to the CICS-key task-lifetime storage of other tasks (CDSA or ECDSA)
- Read access to CICS code
- Read access to CICS control blocks that are accessible by the CICS API

They do not have any access to user-key task-lifetime storage of other tasks.

The defaults for new transaction resource definition attributes specify that existing application programs operate with transaction isolation (the default for the ISOLATE option is YES). Existing applications should run unmodified provided they conform to transaction isolation requirements.

However, a minority of applications may need special definition if they:

- Issue MVS macros directly, or
- Modify CICS control blocks, or
- Have a legitimate need for one task to access, or share, another task's storage

Some existing transactions may share task-lifetime storage in various ways, and this sharing may prevent them running isolated from each other. To allow such transactions to continue to execute, a single common subspace is provided in which all such transactions can execute. They are then isolated from the other transactions in the system that are running in their own subspaces, but able to share each other's data within the common subspace. See "The common subspace and shared storage" for more information.

CICS-key programs execute in the base space and so have read/write access to all CICS-key storage and user-key storage.

The common subspace and shared storage

You might have some transactions where the application programs access one another's storage in a valid way. One such case is when a task waits on one or more event control blocks (ECBs) that are later posted, either by an MVS POST or "hand posting", by another task.

For example, a task can pass the address of a piece of its own storage to another task (by a temporary storage queue or some other method) and then WAIT for the other task to post an ECB to say that it has updated the storage. Clearly, if the original task is executing in a unique subspace, the posting task fails when attempting the update and to post the ECB, unless the posting task is executing in CICS key.

CICS supports the following methods to ensure that transactions that need to share storage can continue to work in the subspace-group environment:

- You can specify that all the related transactions are to run in the common subspace. The common subspace allows tasks that need to share storage to coexist, while isolating them from other transactions in the system. Transactions assigned to the common subspace have the following characteristics:
 - They have read and write access to each other's task-lifetime storage.
 - They have no access of any kind to storage of transactions that run in unique subspaces.
 - They have read access only to CICS storage.

Any group of related transactions that work in user key in CICS/ESA 4.1 should work under CICS Transaction Server for z/OS, Version 3 Release 1 if defined with ISOLATE(NO) to ensure they run in the common subspace. This provides

support for migration, allowing the separation of transactions into their own unique subspaces to be staged gradually after installing CICS and related support.

- You can ensure that the common storage is in SHARED storage by obtaining the storage with the SHARED option.
- You can ensure that the application programs of the transactions that are sharing storage are all defined with EXECKEY(CICS). This ensures that their programs execute in base space, where they have read/write access to all storage. However, this method is not recommended because it does not give any storage protection.

I

You can use the CICS Interdependency Analyzer to help you identify transactions that include the commands such as WAIT EVENT, WAITCICS, WAIT EXTERNAL, and MVS POST. See the *CICS Interdependency Analyzer for z/OS User's Guide and Reference* manual for more information about this utility.

Chapter 47. Transient data control

This chapter describes the three different transient data queues in CICS and also explains automatic transaction initiation.

The CICS transient data control facility provides a generalized queuing facility. Data can be queued (stored) for subsequent internal or external processing. Selected data, specified in the application program, can be routed to or from predefined symbolic transient data queues: either **intrapartition** or **extrapartition**.

Transient data queues are intrapartition if they are associated with a facility allocated to the CICS region, and extrapartition if the data is directed to a destination that is external to the CICS region. Transient data queues must be defined and installed before first reference by an application program.

You can:

- Write data to a transient data queue (WRITEQ TD command)
- Read data from a transient data queue (READQ TD command)
- Delete an intrapartition transient data queue (DELETEQ TD command)

If the TD keyword is omitted, the command is assumed to be for temporary storage. (See Chapter 48, “Temporary storage control,” on page 597 for more information about temporary storage.)

Java and C++

The application programming interface described in this chapter is the EXEC CICS API, which is not used in Java programs. For information about Java programs using the JCICS classes to access transient data services, see the *Java Applications in CICS* component of the CICS Information Center and the JCICS Javadoc html documentation. For information about C++ programs using the CICS C++ classes, see the *CICS C++ OO Class Libraries* manual.

This chapter describes:

- “Intrapartition transient data queues”
- “Extrapartition queues” on page 594
- “Indirect queues” on page 595
- “Automatic transaction initiation (ATI)” on page 595

Intrapartition transient data queues

“Intrapartition” refers to data on direct-access storage devices for use with one or more programs running as separate tasks. Data directed to or from these internal queues is referred to as intrapartition data; it must consist of variable-length records. All intrapartition transient data destinations are held as queues in the same VSAM data set, which is managed by CICS. An intrapartition destination requires a resource definition containing information that locates the queue in the intrapartition data set. Intrapartition queues can be associated with either a terminal or an output data set. When data is written to the queue by a user task, the queue can be used subsequently as input data by other tasks within the CICS region. All access is sequential, governed by read and write pointers. Once a record has been read, it cannot be read subsequently by another task. Intrapartition data may ultimately be transmitted upon request to the terminal or retrieved sequentially from the output data set.

Typical uses of intrapartition data include:

- Message switching
- Broadcasting
- Database access
- Routing of output to several terminals (for example, for order distribution)
- Queuing of data (for example, for assignment of order numbers or priority by arrival)
- Data collection (for example, for batched input from 2780 Data Transmission Terminals)

There are three types of intrapartition transient data queue:

- **Non-recoverable** Non-recoverable intrapartition transient data queues are recovered only on a warm start of CICS. If a unit of work (UOW) updates a non-recoverable intrapartition queue and subsequently backs out the updates, the updates made to the queue **are not** backed out.
- **Physically recoverable** Physically recoverable intrapartition transient data queues are recovered on warm and emergency restarts. If a UOW updates a physically recoverable intrapartition queue and subsequently backs out the updates, the updates made to the queue **are not** backed out.
- **Logically recoverable** Logically recoverable intrapartition transient data queues are recovered on warm and emergency restarts. If a UOW updates a logically recoverable intrapartition queue and subsequently backs out the changes it has made, the changes made to the queue are also backed out. On a warm or an emergency restart, the committed state of a logically recoverable intrapartition queue is recovered. In-flight UOWs are ignored.

If an application is trying to issue a read, write, or delete request and suffers an indoubt failure, it may receive a LOCKED response if WAIT(YES) and WAITACTION(REJECT) are specified in the queue definition.

Extrapartition queues

Extrapartition queues (data sets) reside on any sequential device (DASD, tape, printer, and so on) that are accessible by programs outside (or within) the CICS region. In general, sequential extrapartition queues are used for storing and retrieving data outside the CICS region. For example, one task may read data from a remote terminal, edit the data, and write the results to a data set for subsequent processing in another region. Logging data, statistics, and transaction error messages are examples of data that can be written to extrapartition queues. In general, extrapartition data created by CICS is intended for subsequent batched input to non-CICS programs. Data can also be routed to an output device such as a printer.

Data directed to or from an external destination is referred to as extrapartition data and consists of sequential records that are fixed-length or variable-length, blocked or unblocked. The record format for an extrapartition destination must be defined in a TDQUEUE resource definition by the system programmer. See the *CICS Resource Definition Guide* for details about queue definitions.

If you create a data set definition for the extrapartition queue using JCL, the DD statement for the data set should not include the FREE=CLOSE operand. If FREE=CLOSE is specified, attempts to read the queue after the queue has been closed and then re-opened can receive an IOERR condition. The *CICS System Definition Guide* has more information about defining data sets to CICS.

Indirect queues

Intrapartition and extrapartition queues can be used as indirect queues. Indirect queues provide some flexibility in program maintenance in that data can be routed to one of several queues with only the transient data definition, and not the program itself, having to be changed.

When a transient data definition has been changed, application programs continue to route data to the queue using the original symbolic name; however, this name is now an indirect queue that refers to the new symbolic name. Because indirect queues are established by using transient data resource definitions, the application programmer does not usually have to be concerned with how this is done. Further information about transient data resource definition is in the *CICS Resource Definition Guide*.

Automatic transaction initiation (ATI)

For intrapartition queues, CICS provides the option of automatic transaction initiation (ATI).

A basis for ATI is established by the system programmer by specifying a nonzero trigger level for a particular intrapartition destination. (See the *CICS Resource Definition Guide* for more information about trigger levels.) When the number of entries (created by WRITEQ TD commands issued by one or more programs) in the queue reaches the specified trigger level, a transaction specified in the definition of the queue is automatically initiated. Control is passed to a program that processes the data in the queue; the program must issue repetitive READQ TD commands to deplete the queue.

When the queue has been emptied, a new ATI cycle begins. That is, a new task is scheduled for initiation when the specified trigger level is again reached, whether or not execution of the earlier task has ended. The exact point at which a new ATI cycle begins depends on whether or not the queue is defined as logically recoverable. If the queue is defined with a recoverability attribute (RECOVSTATUS) of No or Physical, the new ATI cycle begins when the queue is read to QZERO. But if the queue is defined with a recoverability attribute of Logical, the new ATI cycle begins only after the task terminates after having read the queue to QZERO.

If an automatically initiated task does not empty the queue, access to the queue is not inhibited. The task may be normally or abnormally ended before the queue is emptied (that is, before a QZERO condition occurs in response to a READQ TD command). If the contents of the queue are to be sent to a terminal, and the previous task completed normally, the fact that QZERO has not been reached means that trigger processing has not been reset and the same task is reinitiated. A subsequent WRITEQ TD command does not trigger a new task if trigger processing has not been reset.

If the contents of the queue are to be sent to a file, the termination of the task has the same effect as QZERO (that is, trigger processing is reset). The next WRITEQ TD command initiates the trigger transaction (if the trigger level has been reached).

If the trigger level of a queue is zero, no task is automatically initiated.

If a queue is logically recoverable, initiation of the trigger transaction is deferred until the next syncpoint.

If the trigger level has already been exceeded because the last triggered transaction abended before clearing the queue, or because the transaction was never started because the MXT limit was reached, another task is not scheduled. This is because QZERO has not been raised to reset trigger processing. If the contents of a queue are destined for a file, the termination of the task resets trigger processing and means that the next WRITEQ TD command triggers a new task.

To ensure that an automatically initiated task completes when the queue is empty, the application program should test for a QZERO condition in preference to some other application-dependent factor (such as an anticipated number of records). Only the QZERO condition indicates an emptied queue.

If the contents of a queue are to be sent to another system, the session name is held in EIBTERMID. If a transaction (started with a destination of system) abends, a new transaction is started in the same way as a terminal.

If you use ATI with a transient data trigger mechanism, it could create inter-transaction affinities that adversely affect your ability to perform dynamic transaction routing. See Chapter 15, “Affinity,” on page 221 for more information about transaction affinity.

A trigger transaction is shunted if it suffers from an indoubt failure. Another trigger transaction is not attached until the shunted UOW commits or backs out the changes it has made following resynchronization.

Chapter 48. Temporary storage control

The CICS temporary storage control facility provides the application programmer with the ability to store data in temporary storage queues, either in main storage, in auxiliary storage on a direct-access storage device, or in a temporary storage data sharing pool. Data stored in a temporary storage queue is known as temporary data.

Java and C++

The application programming interface described in this chapter is the EXEC CICS API, which is not used in Java programs. For information about Java programs using the JCICS classes to access temporary storage services, see *Java Applications in CICS* in the CICS Information Center and the JCICS Javadoc html documentation. For information about C++ programs using the CICS C++ classes, see the *CICS C++ OO Class Libraries* manual.

You can:

- Write data to a temporary storage queue (WRITEQ TS command).
- Update data in a temporary storage queue (WRITEQ TS REWRITE command).
- Read data from a temporary storage queue (READQ TS command).
- Read the next data from a temporary storage queue (READQ TS NEXT command).
- Delete a temporary storage queue (DELETEQ TS command).

The TS keyword may be omitted; temporary storage is assumed if it is not specified.

Exception conditions that occur during execution of a temporary storage control command are handled as described in Chapter 17, “Dealing with exception conditions,” on page 261.

If you use these commands, you could create inter-transaction affinities that adversely affect your ability to perform dynamic transaction routing.

To help you identify potential problems with programs that issue these commands, you can use the scanner and collector components of the CICS Interdependency Analyzer. See the *CICS Interdependency Analyzer for z/OS User's Guide and Reference* for more information about this utility and Chapter 15, “Affinity,” on page 221 for more information about transaction affinity.

This chapter describes:

- “Temporary storage queues”
- “Typical uses of temporary storage control” on page 598

Temporary storage queues

Temporary storage queues are identified by symbolic names that may be up to 16 characters, assigned by the originating task. Temporary data can be retrieved by the originating task or by any other task using the symbolic name assigned to it. To avoid conflicts caused by duplicate names, a naming convention should be established; for example, the operator identifier or terminal identifier could be used

as a suffix to each programmer-supplied symbolic name. Specific items (logical records) within a queue are referred to by relative position numbers.

Temporary storage queues remain intact until they are deleted by the originating task, by any other task, or by an initial or cold start; before deletion, they can be accessed any number of times. Even after the originating task is terminated, temporary data can be accessed by other tasks through references to the symbolic name under which it is stored.

Temporary data can be stored either in main storage or in auxiliary storage. Generally, main storage should be used if the data is needed for short periods of time; auxiliary storage should be used if the data is to be kept for long periods of time. Data stored in auxiliary storage is retained after CICS termination and can be recovered in a subsequent restart, but data in main storage cannot be recovered. Main storage might be used to pass data from task to task, or for unique storage that allows programs to meet the requirement of CICS that they be quasi-reentrant (that is, serially reusable between entry and exit points of the program).

Temporary storage data sharing provides another type of temporary storage queue that can be supported concurrently. The temporary storage queues can be defined as local, remote, or shared, and they can be stored in temporary storage pools in the coupling facility.

Typical uses of temporary storage control

A temporary storage queue that has only one record can be treated as a single unit of data that can be accessed using its symbolic name. Using temporary storage control in this way provides a typical scratch-pad capability. This type of storage should be accessed using the READQ TS command with the ITEM option; not doing so may cause the ITEMERR condition to be raised.

In general, temporary storage queues of more than one record should be used only when direct access or repeated access to records is necessary; transient data control provides facilities for efficient handling of sequential data sets.

Some uses of temporary storage queues are:

Terminal paging

A task could retrieve a large master record from a direct-access data set, format it into several screen images (using BMS), store the screen images temporarily in auxiliary storage, and then ask the terminal operator which “page” (screen image) is desired. The application programmer can provide a program (as a generalized routine or unique to a single application) to advance page by page, advance or back up a relative number of pages, and so on.

A suspend data set

Suppose a data collection task is in progress at a terminal. The task reads one or more units of input and then allows the terminal operator to interrupt the process by some kind of coded input. If not interrupted, the task repeats the data collection process. If interrupted, the task writes its incomplete data to temporary storage and terminates. The terminal is now free to process a different transaction (perhaps a high-priority inquiry). When the terminal is available to continue data collection, the operator initiates the task in a “resume” mode, causing the task to recall its suspended data from temporary storage and continue as though it had not been interrupted.

Preprinted forms

An application program can accept data to be written as output on a preprinted form. This data can be stored in temporary storage as it arrives. When all the data has been stored, it can first be validated and then transmitted in the order required by the format of the preprinted form.

Part 8. Testing and debugging applications

	Chapter 49. Testing applications	603
	Preparing the application for testing	604
	Preparing the system for testing	604
	Chapter 50. Execution diagnostic facility (EDF)	607
	Restrictions when using EDF	608
#	OPEN TCBs and EDF	610
	Parameter list stacking	610
	Security considerations	610
	What does EDF display?	611
	The header	611
	The body	612
	At program initiation	612
	At the start of execution of a CICS command	612
	At the end of execution of a command	614
	At program and task termination	616
	At abnormal termination	617
	Using EDF	618
	Interrupting program execution	619
	Using EDF in single-screen mode	620
	Checking pseudoconversational programs	621
	Using EDF in dual-screen mode	622
	EDF and remote transactions	622
	EDF and non-terminal transactions	622
	EDF and DTP programs	623
	EDF and distributed program link commands	623
	Stopping EDF	624
	Overtyping to make changes	624
	EDF responses	626
	Using EDF menu functions	626
	Chapter 51. Temporary storage browse (CEBR)	633
	Using the CEBR transaction	633
	What does the CEBR transaction display?	635
	The header	635
	The command area	635
	The body	635
	The message line	635
	Using the CEBR function keys	636
	Using the CEBR commands	637
	Using the CEBR transaction with transient data	639
	Chapter 52. Command-level interpreter (CECI)	641
	What does CECI display?	641
	The command line	641
	The status line	642
	Command syntax check	643
	About to execute command	644
	Command execution complete	645
	The body	646
	The message line	646
	CECI options on function keys	646
	Using CECI	648

	Making changes	649
	Using the CECI function keys	650
	Expanded area	650
	Variables.	650
	Defining variables	651
	The EXEC interface block (EIB)	652
	Error messages display	652
	Saving commands	653
	How CECI runs	654
	CECI sessions	654
	Abends	654
	Exception conditions	655
	Program control commands.	655
	Terminal sharing	655
#	Shared storage: ENQ commands without LENGTH option	655
	Chapter 53. Using debuggers with CICS applications	657
	Debugging profiles	658
	Using debugging profiles to select programs for debugging	659
	Using generic parameters in debugging profiles	661
	Chapter 54. Debugging CICS applications from a workstation.	663
	Preparing to debug applications from a workstation	663
	Chapter 55. Using Debug Tool with CICS applications.	665
	About Debug Tool	665
	Preparing to debug applications with Debug Tool	665

Chapter 49. Testing applications

Java

This guidance does not relate to testing Java applications.

You can use the following methods to test CICS application programs:

Single-thread testing

A **single-thread** test takes one application transaction at a time, in an otherwise “empty” CICS system, and sees how it behaves. This enables you to test the program logic, and also shows whether or not the basic CICS information (such as resource definition) is correct. It is quite feasible to test this single application in one CICS region while your normal, online production CICS system is active in another.

Multithread testing

A **multithread** test involves several concurrently active transactions. Naturally, all the transactions are in the same CICS region, so you can readily test the ability of a new transaction to coexist with them.

You may find that a transaction that works perfectly in its single-thread testing still fails in the multithread test. It may also cause other transactions to fail, or even terminate CICS.

Regression testing

A **regression** test is used to make sure that all the transactions in a system continue to do their processing in the same way both before and after changes are applied to the system. This is to ensure that fixes applied to solve one problem do not cause further problems. It is a good idea to build a set of miniature files to perform your tests on, because it is much easier to examine a small data file for changes.

A good regression test exercises all the code in every program; that is, it explores all tests and possible conditions. As your system develops to include more transactions, more possible conditions, and so on, add these to your test system to keep it in step. The results of each test should match those from the previous round of testing. Any discrepancies are grounds for suspicion. You can compare terminal output, file changes, and log entries for validity.

Sequential terminal support (described in “Using sequential terminal support” on page 381), can be useful for regression testing. When you have a module that has worked for some time and is now being modified, you need to rerun your old tests to ensure that the function still works. Sequential terminal support makes it easy to maintain a “library” of old test cases and to rerun them when needed.

Sequential terminal support allows you to test programs without having to use a telecommunication device. System programmers can specify that sequential devices be used as terminals (using the terminal control table (TCT)). These sequential devices may be card readers, line printers, disk units, or magnetic tape units. They can also be combinations of sequential devices such as:

- A card reader and line printer (CRLP)
- One or more disk or tape data sets as input
- One or more disk or tape data sets as output

You can prepare a stream of transaction test cases to do the basic testing of a program module. As the testing progresses, you can generate additional

transaction streams to validate the multiprogramming capabilities of the programs or to allow transaction test cases to run concurrently.

You have to do two main tasks before you can test and debug an application:

1. "Preparing the application for testing"
2. "Preparing the system for testing"

Preparing the application for testing

To prepare the application and system table entries you should do the following:

1. Translate, assemble or compile, and link-edit each program. Make sure that there are no error messages on any of these three steps for any program before you begin testing.
2. Use the DEBUG and EDF options on your translator step, so that you can use translator statement numbers with execution diagnostic facility (EDF) displays.
3. Use the COBOL compiler options CLIST and DMAP so that you can relate storage locations in dumps and EDF displays to the original COBOL source statements, and find your variables in working storage.
4. Use the RDO DEFINE PROFILE command to generate a profile for your transactions to use, and make sure the definitions are INSTALLED.
5. Use the RDO DEFINE TRANSACTION command for each transaction in your application, and make sure the definitions are INSTALLED.
6. If your system does not use program autoinstall, use the RDO DEFINE PROGRAM command for each program used in the application, and make sure the definitions are INSTALLED.
7. If your system does not use program autoinstall, use the RDO DEFINE MAPSET command for each map set in the application, and make sure each definition is INSTALLED.
8. Use the RDO DEFINE FILE command, or put an entry in the FCT, for each file used. If you use RDO, make sure the definitions are INSTALLED.
9. Build at least a test version of each of the files required.
10. Define each of the transient data destinations to be used by the application.
11. Put job control DD cards in the startup job stream for each file used in the application.
12. Prepare some test data.

Preparing the system for testing

To prepare the system for debugging you should do the following:

1. Make sure that EDF is available in your system, by including group DFHEDF in the list you specify in the GRPLIST system initialization
2. Set up appropriate tracing options for your application. For details about setting up tracing options, see the *CICS Problem Determination Guide*.
3. Make sure that transaction dumping is enabled for all transaction dump codes, and that system dumping is enabled for all system dump codes. These are, anyway, the default settings. For information about setting up dump options, see the *CICS Problem Determination Guide*.
4. Be prepared to print the dumps. Have a DFHDU640 job stream or procedure ready, and have the CICS dump data sets defined in your startup procedure.
5. Contact your system programmer for information about SDUMP data sets available on your system and access to JCL for processing them.

6. Enable CICS to detect loops, by setting the ICVR parameter in the SIT to a number greater than zero. Something between five and ten seconds (ICVR=5000 to ICVR=10000) is usually a workable value.
7. Generate statistics. For more information about using statistics, see the *CICS Performance Guide*.

Chapter 50. Execution diagnostic facility (EDF)

You can use the execution diagnostic facility (EDF) to test an application program online, without modifying the program or the program-preparation procedure. The CICS execution diagnostic facility is supported by the CICS-supplied transaction, CEDF, which invokes the DFHEDFP program.

Note: You can also invoke CEDF indirectly through another CICS-supplied transaction, CEDX, which enables you to specify the name of the transaction you want to debug. When this chapter refers to the CEDF transaction (for example, when it explains about CICS starting a new CEDF task below) remember that it may have been invoked by the CEDX command.

The names of your programs should not begin with the letters “DFH” because this prefix is used for CICS system modules and samples. Attempting to use EDF on a CICS-supplied transaction has no effect. However, you can use EDF with CICS sample programs and some user-replaceable modules. (For example, you can use EDF to debug DFHPEP.)

EDF intercepts the execution of CICS commands in the application program at various points, allowing you to see what is happening. Each command is displayed before execution, and most are displayed after execution is complete. Screens sent by the application program are preserved, so you can converse with the application program during testing, just as a user would on a production system.

When a transaction runs under EDF control, EDF intercepts it at the following points, allowing you to interact with it:

- At **program initiation**, after the EXEC interface block (EIB) has been updated, but before the program is given control.
- At the **start of the execution of each CICS command**. This interrupt happens after the initial trace entry has been made, but before the command has been performed. Both standard CICS commands and the Front End Programming Interface (FEPI) commands are intercepted. EXEC DLI and EXEC SQL commands and any requests processed through the resource manager interface are also intercepted at this point.
- At the **end of the execution of every command** except for ABEND, XCTL, and RETURN commands (although these commands could raise an error condition that EDF displays). EDF intercepts the transaction when it finishes processing the command, but before the HANDLE CONDITION mechanism is invoked, and before the response trace entry is made.
- At **program termination**.
- At **normal task termination**.
- When an **ABEND** occurs and after **abnormal task termination**.

If you want to work through an example of EDF, see *Designing and Programming CICS Applications*, which guides you through a sample EDF session.

Note: For a program translated with the option NOEDF, these intercept points still apply, apart from before and after the execution of each command. For a program with CEDF defined as NO on its resource definition or by the program autoinstall exit, the program initiation and termination screens are suppressed as well.

Each time EDF interrupts the execution of the application program a new CEDF task is started. Each CEDF task is short lived, lasting only long enough for the appropriate display to be processed.

The terminal that you are using for the EDF interaction must be in transceive (ATI/TTI) status and be able to send and receive data. This is the most common status for display terminals, but you can find out by asking your system programmer to check its status, or you can use CEMT.

For a transaction initiated at a terminal, you can use EDF on the same terminal as the transaction you are testing, or on a different one. On the same terminal, you **must** start by clearing the screen and entering the transaction code CEDF, otherwise you may get unpredictable results. The message **THIS TERMINAL: EDF MODE ON** is displayed at the top of an empty screen. You clear the screen again and run your transaction in the normal way.

When you are using EDF, the user task is not directly purgable. If you need to terminate the task, first forcepurge the CEDF task, then attempt to press the Enter key while the EDF screen is displayed. If pressing the Enter key brings no response, forcepurge the CEDF task a second time. CEDF will terminate, and the user transaction will receive an AED3 abend.

This chapter describes:

- “Restrictions when using EDF”
- “What does EDF display?” on page 611
- “Using EDF” on page 618
- “Overtyping to make changes” on page 624
- “Using EDF menu functions” on page 626

Restrictions when using EDF

User application programs that are to be debugged using EDF must be assembled (or compiled) with the translator option EDF, which is the default. If you specify NOEDF, the program cannot be debugged using EDF. There is no performance advantage in specifying NOEDF, but the option can be useful to prevent commands in well debugged subprograms appearing on EDF displays.

Application programs that are to be debugged using EDF must also have the
attribute CEDF(YES) in their resource definition, which is the default. If a program is
defined with CEDF(YES) and compiled with the translator option EDF, EDF
diagnostic screens are displayed for the program. If the program is defined with
CEDF(YES) but compiled with the translator option NOEDF, only the program
initiation and termination screens are displayed. If CEDF(NO) is specified, no EDF
screens are displayed.

If a program with the attribute CEDF(NO) links to a program with the attribute
CEDF(YES), it might not be possible to use EDF for the transaction. For example, if
the CICSplex SM dynamic transaction routing program EYU9XLOP is defined with
the attribute CEDF(NO), and the user-replaceable program EYU9WRAM (for
workload management processing) is defined with the attribute CEDF(YES), you
cannot use EDF to debug EYU9WRAM. For successful debugging of multiple
programs within a transaction, ensure that all the programs are defined with
CEDF(YES).

There are some restrictions on the use of EDF that make it preferable or even necessary to use one particular screen mode:

- EDF can be used only in single-screen mode when running a remote transaction.
- VM PASSTHRU is not supported by EDF when testing in single-screen mode.
- In single-screen mode, neither the user transaction nor CEDF should specify message journaling, because the messages interfere with the EDF displays. Message journaling is controlled by the profile definition for each transaction.
- In single screen mode, the CEDF transaction should not specify PROTECT=YES in its profile definition. If this option is specified, message protection for the CEDF transaction is ignored. The user transaction can still specify the PROTECT=YES option even when running under CEDF. This restriction does not apply to dual-screen mode.
- If a SEND LAST command is issued, EDF is ended before the command is processed if you are using single-screen mode.
- If you want to test an application program that uses screen partitions, or that does its own request unit (RU) chaining, you must run in dual-screen mode.
- In single-screen mode, if the profile for the user transaction specifies INBFMH=ALL or INBFMH=DIP, the profile for CEDF must have the same INBFMH value. Otherwise the user transaction abends ADIR. Dual-screen mode does not require the profiles to match in this respect.
- If the inbound reply mode is set to “character” to enable the attribute setting keys, EDF disables the keys in single-screen mode.
- When using CECI under EDF in dual-screen mode, you should be aware that certain commands (for example, ASSIGN and ADDRESS) are issued against the EDF terminal and not the transaction terminal. See “INVOKE CECI” on page 627 for information about how to invoke CECI from CEDF.
- TCAM terminals are supported by EDF, but only in dual-screen mode, and provided that the terminals are not pooled.

Note: In CICS TS 3.1, local TCAM terminals are not supported. The only TCAM terminals supported are remote terminals connected to a pre-CICS TS 3.1 terminal-owning region by the DCB (not ACB) interface of TCAM.

- When using EDF in dual-screen mode, you should avoid starting a second task at the EDF terminal, for example by issuing a START command. Because EDF is a pseudoconversational transaction, it does not prevent a second task from starting at the terminal it is using. This may lead to a deadlock in certain circumstances.
- When using EDF screen suppression in dual screen mode, commands that cause a long wait, such as DELAY, WAIT, or a second RECEIVE, may cause EDF to appear as if it had finished. If the task is ABENDED, EDF is reactivated at the monitoring terminal.

Other restrictions apply to both screen modes:

- If a transaction issues the FREE command, EDF is switched off without warning.
- To test a user transaction executing on a remote CICS at a release level earlier than CICS/ESA 3.1.1, you must run the transaction under control of CRTE, as explained in “EDF and remote transactions” on page 622.
- EDF does not intercept calls to the CPI Communications interface (CPI-C) or the SAA Resource Recovery interface (CPI-RR). You can test transactions that use CPI calls under EDF, but you cannot see EDF displays at the call points.
- When processing a SIGNON command, CEDF suppresses display of the password value to reduce the risk of accidental disclosure.

OPEN TCBs and EDF

Even if your program would normally run using an OPEN TCB (L8, L9, X8, or X9)
CEDF forces the program to run on the QR TCB, because CEDF itself is not
threadsafe.

Parameter list stacking

CEDF only has one level of stacking for its copies of the EXEC CICS parameter list. This means that if an application calls an EXEC-capable global user exit or user-replaceable module (URM), the parameter list for the EXEC CICS commands issued by the global user exit or URM may overlay the parameter list for EXEC CICS commands issued by the main program.

Security considerations

EDF is such a powerful tool that your installation may restrict its use with attach-time security. (The external security manager used by your installation defines the security attributes for the EDF transaction.) If this has been done, and you are not authorized to use CEDF, you cannot initiate the transaction.

For guidance on using security, see your system programmer or the *CICS RACF Security Guide*.

What does EDF display?

All EDF displays have the same general format, but the contents depend on the point at which the task was interrupted. The display indicates which of these interception points has been reached and shows information relevant to that point. Figure 134 shows a typical display; occurring after execution of a SEND MAP command.

```
TRANSACTION: AC20 PROGRAM: DFH0VT1 TASK: 00032 APPLID: 1234567 DISPLAY:00
STATUS:  COMMAND EXECUTION COMPLETE 1
EXEC CICS SEND MAP
  MAP ('T1      ')
  FROM ('.....'.....')
  LENGTH (154)
  MAPSET ('DFH0T1 ')
  CURSOR 2
  TERMINAL
  ERASE
  NOFLUSH
  NOHANDLE

  OFFSET:X'002522'  LINE:00673  EIBFN=X'1804'
  RESPONSE: NORMAL  EIBRESP=0 3
ENTER:  CONTINUE 4
PF1 : UNDEFINED      PF2 : SWITCH HEX/CHAR  PF3 : END EDF SESSION
PF4 : SUPPRESS DISPLAYS PF5 : WORKING STORAGE  PF6 : USER DISPLAY
PF7 : SCROLL BACK     PF8 : SCROLL FORWARD  PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY PF11: EIB DISPLAY    PF12: ABEND USER TASK
```

Figure 134. Typical EDF display

Note: 1 Header 2 Body 3 Message line 4 Menu of functions

The display consists of a header, a body (the primary display area), a message line, and a menu of functions you can select at this point. If the body does not fit on one screen, EDF creates multiple screens, which you can scroll through using PF7 and PF8. The header, menu, and message areas are repeated on each screen.

The header

The header shows:

- The identifier of the transaction being executed
- The name of the program being executed
- The internal task number assigned by CICS to the transaction
- The applid of the CICS region where the transaction is being executed
- A display number
- STATUS, that is, the reason for the interception by EDF

The body

The body or main part of the display contains the information that varies with the point of intercept. The following screens show the body contents:

- “At program initiation”
- “At the start of execution of a CICS command”
- “At the end of execution of a command” on page 614
- “At program and task termination” on page 616
- “At abnormal termination” on page 617

At program initiation

At **program initiation**, as shown in Figure 135, EDF displays the COMMAREA (if any) and the contents of the principal fields in the EIB. For programming information about these EIB fields, see the *CICS Application Programming Reference* manual. If there isn't a COMMAREA, line 4 on the screen is left blank and EIBCALEN has a value of zero.

```
TRANSACTION: AC20 PROGRAM: DFH0VT1 TASK: 00032 APPLID: 1234567 DISPLAY:00
STATUS: PROGRAM INITIATION

COMMAREA      = '3476559873'
EIBTIME       = 92920
EIBDATE       = 91163
EIBTRNID      = 'AC20'
EIBTASKN      = 32
EIBTRMID      = 'S246'

EIBCPOSN      = 4
EIBCALEN      = 10
EIBRID        = X'7D'
EIBFN         = X'0000'
EIBRCODE      = X'000000000000'
EIBDS         = '.....'
+ EIBREQID     = '.....'

ENTER: CONTINUE
PF1 : UNDEFINED      PF2 : SWITCH HEX/CHAR    PF3 : END EDF SESSION
PF4 : SUPPRESS DISPLAYS PF5 : WORKING STORAGE    PF6 : USER DISPLAY
PF7 : SCROLL BACK     PF8 : SCROLL FORWARD    PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY PF11: EIB DISPLAY       PF12: UNDEFINED
```

Figure 135. Typical EDF display at program initiation

At the start of execution of a CICS command

At the **start of execution of a CICS command**, EDF displays the command, including keywords, options, and argument values, as shown in Figure 136 on page 613. You can display the information in hexadecimal or character form (and switch from one to the other) by pressing PF2. If character format is requested, numeric arguments are shown in signed numeric character format.

```

TRANSACTION: AC20 PROGRAM: DFH0VT1 TASK: 00032 APPLID: 1234567 DISPLAY:00
STATUS: ABOUT TO EXECUTE COMMAND
EXEC CICS SEND MAP
  MAP ('T1      ')
  FROM ('.....'..')
  LENGTH (154)
  MAPSET ('DFH0T1 ')
  CURSOR
  TERMINAL
  ERASE
  NOFLUSH
  NOHANDLE

OFFSET:X'002522'  LINE:00673  EIBFN=X'1804'

ENTER: CONTINUE
PF1 : UNDEFINED          PF2 : SWITCH HEX/CHAR      PF3 : UNDEFINED
PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE    PF6 : USER DISPLAY
PF7 : SCROLL BACK        PF8 : SCROLL FORWARD    PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY    PF11: EIB DISPLAY        PF12: ABEND USER TASK

```

Figure 136. Typical EDF display at start of execution of a CICS command

Figure 137 shows a similar screen for the **start of execution** of an EXEC SQL command running with DB2 version 2.3.

```

TRANSACTION: LOKO PROGRAM: TLOKO TASK: 00082 APPLID: 1234567 DISPLAY:00
STATUS: ABOUT TO EXECUTE COMMAND
CALL TO RESOURCE MANAGER DSNCSQL
EXEC SQL UPDATE
  DBRM=TLOK0, STMT=00242, SECT=00001
  IVAR 001: TYPE=CHAR, LEN=00010 AT X'001E5A99'
  DATA=X'F0F0F0F0F0F1F0F0F0F0'

OFFSET:X'000298'  LINE: UNKNOWN EIBFN= X'0A02'
ENTER: CONTINUE
PF1 : UNDEFINED          PF2 : UNDEFINED          PF3 : UNDEFINED
PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE    PF6 : USER DISPLAY
PF7 : SCROLL BACK        PF8 : SCROLL FORWARD    PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY    PF11: EIB DISPLAY        PF12: ABEND USER TASK

```

Figure 137. Typical SQL display at start of execution of a SQL command

In addition to options and values, the command is identified by its hexadecimal offset within the program. If the program was translated with the DEBUG translator option, the line number also appears, as shown in Figure 136. (See “Defining translator options” on page 74 for information about this option.)

At the start of an EXEC SQL or EXEC DLI command, the body of the EDF display shows you the parameter list of the CALL to which your command translates. If a DLI command generates multiple CALL statements, you see only the last CALL statement.

At the end of execution of a command

At the **end of execution of a command**, EDF provides a display in the same format as at the start of the command. At this point, you can see the effects of executing the command, in the values of the variables returned or changed and in the response code. EDF does not provide this display for the ABEND, XCTL, and RETURN commands (although these commands could raise an error condition that EDF displays). The completion screen corresponding to the **about to execute** screen in Figure 136 on page 613 is shown in Figure 138.

```
TRANSACTION: AC20 PROGRAM: DFH0VT1 TASK: 00054 APPLID: 1234567 DISPLAY:00
STATUS: COMMAND EXECUTION COMPLETE
EXEC CICS SEND MAP
MAP ('T1      ')
FROM ('.....'.....')
LENGTH (154)
MAPSET ('DFH0T1 ')
CURSOR
TERMINAL
ERASE
NOFLUSH
NOHANDLE

OFFSET:X'002522'   LINE:00673           EIBFN=X'1804'
RESPONSE: NORMAL           EIBRESP=0

ENTER: CONTINUE
PF1 : UNDEFINED          PF2 : SWITCH HEX/CHAR      PF3 : END EDF SESSION
PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE      PF6 : USER DISPLAY
PF7 : SCROLL BACK        PF8 : SCROLL FORWARD      PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY    PF11: EIB DISPLAY          PF12: ABEND USER TASK
```

Figure 138. Typical EDF display at completion of a CICS command

For CICS commands, response codes are described both by name (for example, NORMAL or NOTFND) and by the corresponding EIBRESP value in decimal form. For DL/I, the response code is a 2-character DL/I status code, and there is no EIBRESP value. Programming information, including a list of EIBRESP codes, is in *CICS Application Programming Reference* manual, and DL/I codes are documented in the *Application Programming: EXEC DLI Commands*.

Figure 139 and Figure 140 show typical screens for an EXEC DLI command.

```

TRANSACTION: XDLI PROGRAM: UPDATE TASK: 00111 APPLID: 1234567 DISPLAY: 00
STATUS: COMMAND EXECUTION COMPLETE
EXEC DLI GET NEXT
USING PCB (+00003)
FIRST
SEGMENT ('A      ')
INTO ('          ')
SEGLength (+00012)
FIRST
VARIABLE
+SEGMENT ('B      ')

OFFSET:X'000246'  LINE: 00000510          EIBFN:X'000C'
RESPONSE: 'AD'

ENTER: CONTINUE
PF1 : UNDEFINED          PF2 : SWITCH HEX/CHAR    PF3 : END EDF SESSION
PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE    PF6 : USER DISPLAY
PF7 : SCROLL BACK        PF8 : SCROLL FORWARD    PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY    PF11: EIB DISPLAY      PF12: ABEND USER TASK

```

Figure 139. Typical EDF display at completion of a DLI command (screen one)

```

TRANSACTION: XDLI PROGRAM: UPDATE TASK: 00111 APPLID: 1234567 DISPLAY: 00
STATUS: COMMAND EXECUTION COMPLETE
EXEC DLI GET NEXT
+
FIRST
SEGMENT ('C      ')
SEGLength (+00010)
LOCKED
INTO ('SMITH     ')
WHERE (ACCOUNT = '12345')
FIELDLENGTH (+00005)

OFFSET:X'000246'  LINE: 00000510          EIBFN:X'000C'
RESPONSE: 'AD'

ENTER: CONTINUE
PF1 : UNDEFINED          PF2 : SWITCH HEX/CHAR    PF3 : END EDF SESSION
PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE    PF6 : USER DISPLAY
PF7 : SCROLL BACK        PF8 : SCROLL FORWARD    PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY    PF11: EIB DISPLAY      PF12: ABEND USER TASK

```

Figure 140. Typical EDF display at completion of a DLI command (screen two)

```

TRANSACTION: LOKO PROGRAM: TLOKO TASK: 00111 APPLID: 1234567 DISPLAY: 00
STATUS:  COMMAND EXECUTION COMPLETE
CALL TO RESOURCE MANAGER DSNCSQL
EXEC SQL UPDATE
PLAN=TLOK0,      DBRM=TLOK0,      STMT=00242, SECT=00001
SQL COMMUNICATION AREA:
SQLCABC          = 136                      AT X'001E5A18'
SQLCODE          = 000                      AT X'001E5A1C'
SQLERRML         = 000                      AT X'001E5A20'
SQLERRMC         = ''                      AT X'001E5A22'
SQLERRP          = 'DSN'                   AT X'001E5A68'
SQLERRD(1-6)     = 000, 000, 00001, -1, 00000, 000 AT X'001E5A70'
SQLWARN(0-A)     = ' _ _ _ _ _ '          AT X'001E5A88'
SQLSTATE         = 00000                   AT X'001E5A93'

OFFSET:X'000298'  LINE: UNKNOWN EIBFN= X'0A02'
RESPONSE:

ENTER:  CONTINUE
PF1 : UNDEFINED          PF2 : UNDEFINED          PF3 : END EDF SESSION
PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE      PF6 : USER DISPLAY
PF7 : SCROLL BACK        PF8 : SCROLL FORWARD      PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY    PF11: EIB DISPLAY          PF12: ABEND USER TASK

```

Figure 141. Typical SQL display at completion of an SQL command

At program and task termination

At **program termination** and **normal task termination**, there is no body information; all the pertinent information is in the header. Figure 142 and Figure 143 on page 617 show summarized screens for program and task termination.

```

TRANSACTION: AC20 PROGRAM: DFH0VT1 TASK: 00054 APPLID: 1234567 DISPLAY:00
STATUS:  PROGRAM TERMINATION

ENTER:  CONTINUE
PF1 : UNDEFINED          PF2 : SWITCH HEX/CHAR      PF3 : UNDEFINED
PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE      PF6 : USER DISPLAY
PF7 : SCROLL BACK        PF8 : SCROLL FORWARD      PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY    PF11: EIB DISPLAY          PF12: ABEND USER TASK

```

Figure 142. Typical EDF display at program termination


```
TRANSACTION: AC20          TASK: 00054 APPLID: 1234567 DISPLAY: 00
STATUS:  TASK TERMINATION
```

```
CONTINUE EDF? (ENTER YES OR NO)          REPLY: YES
ENTER:  CONTINUE
PF1 : UNDEFINED          PF2 : SWITCH HEX/CHAR      PF3 : END EDF SESSION
PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE      PF6 : USER DISPLAY
PF7 : SCROLL BACK        PF8 : SCROLL FORWARD      PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY   PF11: EIB DISPLAY          PF12: UNDEFINED
```

Figure 143. Typical EDF display at task termination

At abnormal termination

When an **abend** or **abnormal task termination** occurs, EDF displays the screens shown in Figure 144 and Figure 145 on page 618.

```
TRANSACTION: AC20 PROGRAM: DFH0VT1 TASK:00054 APPLID: 1234567 DISPLAY: 00
STATUS:  AN ABEND HAS OCCURRED
  COMMAREA      = '1287656678'
  EIBTIME       = 135510
  EIBDATE       = 91163
  EIBTRNID      = 'AC20'
  EIBTASKN      = 76
  EIBTRMID      = 'S232'
  EIBCPOSN      = 4
  EIBCALEN      = 10
  EIBRID        = X'7D'
  EIBFN         = X'1804' SEND                      AT X'032F059A'
  EIBRCODE      = X'000000000000'                   AT X'032F059B'
  EIBDS         = '.....'                         AT X'032F059D'
+ EIBREQID      = '.....'

  ABEND :   ABCD

ENTER:  CONTINUE
PF1 : UNDEFINED          PF2 : SWITCH HEX/CHAR      PF3 : END EDF SESSION
PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE      PF6 : USER DISPLAY
PF7 : SCROLL BACK        PF8 : SCROLL FORWARD      PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY   PF11: EIB DISPLAY          PF12: UNDEFINED
```

Figure 144. Typical EDF display when an abend occurs

```

TRANSACTION: AC20                                TASK: 00054 APPLID: 1234567 DISPLAY: 00
STATUS: ABNORMAL TASK TERMINATION
  COMMAREA   = '2934564671'
  EIBTIME    = 135510
  EIBDATE    = 91163
  EIBTRNID   = 'AC20'
  EIBTASKN   = 76
  EIBTRMID   = 'S232'
  EIBCPOSN   = 4
  EIBCALEN   = 10
  EIBAID     = X'7D'                                AT X'032F059A'
  EIBFN      = X'1804' SEND                        AT X'032F059B'
  EIBRCODE    = X'000000000000'                    AT X'032F059D'
  EIBDS       = '.....'
+ EIBREQID    = '.....'

ABEND : ABCD
CONTINUE EDF? (ENTER YES OR NO)                      REPLY: YES
ENTER: CONTINUE
PF1 : UNDEFINED          PF2 : SWITCH HEX/CHAR      PF3 : END EDF SESSION
PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE      PF6 : USER DISPLAY
PF7 : SCROLL BACK        PF8 : SCROLL FORWARD      PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY   PF11: EIB DISPLAY          PF12: UNDEFINED

```

Figure 145. Typical EDF display at abnormal task termination

The body displays the COMMAREA and the values of the fields in the EIB as well as the following items:

- The abend code
- If the abend code is ASRA (that is, a program interrupt has occurred), the program status word (PSW) at the time of interrupt, and the source of the interrupt as indicated by the PSW
- If the PSW indicates that the instruction giving rise to the interrupt is within the application program, the offset of that instruction relative to the main entry point

Using EDF

You can run EDF by invoking either the CEDF or CEDX transaction.

If you are testing a non-terminal transaction, use the CEDX transaction, which enables you to specify the name of the transaction.

If you are testing a transaction that is associated with a terminal, you can run EDF in the following ways:

- “Using EDF in single-screen mode” on page 620
- “Using EDF in dual-screen mode” on page 622
- “EDF and remote transactions” on page 622
- “EDF and non-terminal transactions” on page 622
- “EDF and DTP programs” on page 623

Generally, you can use whichever method you prefer, but there are a few situations in which one or the other is required. You must use single-screen mode for remote transactions. See “Restrictions when using EDF” on page 608 for other conditions which affect your choice.

Interrupting program execution

The power of EDF lies in what you can do at each of the intercept points. For example, you can:

- Change the argument values before a command is executed. For CICS commands, you cannot change the actual command, or add or delete options, but you can change the value associated with any option. You can also suppress execution of the command entirely using NOOP. See page 625 for further details.
- Change the results of a command, either by changing the argument values returned by execution or by modifying the response code. This allows you to test branches of the program that are hard to reach using ordinary test data (for example, what happens on an input/output error). It also allows you to bypass the effects of an error to check whether this eliminates a problem.
- Display the working storage of the program, the EIB, and for DL/I programs, the DIB.
- Invoke the command interpreter (CECI). Under CECI you can execute commands that are not present in the program to gain additional information or change the execution environment.
- Display any other location in the CICS region.
- Change the working storage of the program and most fields in the EIB and the DIB. EDF stops your task from interfering with other tasks by preventing you from changing other areas of storage.
- Display the contents of temporary storage and transient data queues.
- Suppress EDF displays until one or more of a set of specific conditions is fulfilled. This speeds up testing.
- Retrieve up to 10 previous EDF displays or saved screens.
- Switch off EDF mode and run the application normally.
- Abend the task.

The first two types of changes are made by overtyping values in the body of the command displays. “Overtyping to make changes” on page 624 tells you how to do this. You use the function keys in the menu for the others; “Using EDF menu functions” on page 626 tells you exactly what you can do and how to go about it.

```
TRANSACTION: DLID PROGRAM: DLID TASK: 00049 APPLID: IYAHZCIB DISPLAY:00  
ADDRESS: 00000000
```

WORKING STORAGE IS NOT AVAILABLE

ENTER: CURRENT DISPLAY

PF1 : UNDEFINED	PF2 : BROWSE TEMP STORAGE	PF3 : UNDEFINED
PF4 : EIB DISPLAY	PF5 : INVOKE CECI	PF6 : USER DISPLAY
PF7 : SCROLL BACK HALF	PF8 : SCROLL FORWARD HALF	PF9 : UNDEFINED
PF10: SCROLL BACK FULL	PF11: SCROLL FORWARD FULL	PF12: REMEMBER DISPLAY

Figure 146. Typical EDF display from which CECI can be invoked

Using EDF in single-screen mode

When you use EDF with just one terminal, the EDF inputs and outputs are interleaved with those from the transaction. This sounds complicated, but works quite easily in practice. The only noticeable peculiarity is that when a SEND command is followed by a RECEIVE command, the display sent by the SEND command appears twice: once when the SEND is executed, and again when the RECEIVE is executed. It is not necessary to respond to the first display, but if you do, EDF preserves anything that was entered from the first display to the second.

You start EDF by:

- Entering transaction code CEDF from a cleared screen, or
- Pressing the appropriate PF key (if one has been defined for EDF)

Next, you start the transaction to be tested by:

1. Pressing the CLEAR key to clear the screen
2. Entering the transaction code of the transaction you want to test

When both EDF and the user transaction are sharing the same terminal, EDF restores the user transaction display at the following times:

- When the transaction requires input from the operator
- When you change the transaction display
- At the end of the transaction
- When you suppress the EDF displays
- When you request USER DISPLAY

To enable restoration, user displays are remembered at the following times:

1. At start of task, before the first EDF screen for the task is displayed
2. Before the next EDF screen is displayed, if the user display has been changed
3. On leaving SCREEN SUPPRESS mode

If CEDF is used with an application program that has been translated with option NOEDF, or one that has NO specified for CEDF in its resource definition:

- It is not possible for EDF to ascertain when the display is changed by that application program.
- Therefore EDF does not know when to save, for later use, a copy of that display.
- This means that, unless either situation 1 on page 620 or 3 on page 620 also apply, the next EDF display overwrites any display sent by the application program.
- EDF has not saved the application's current display, and is unaware that it has changed.
- The changed (but now overwritten) display cannot be restored.

Similarly, in these circumstances:

- CEDF cannot restore the current display when it is about to be changed by the application, or when the transaction requires input from the operator.
- This means that an output command to the principal facility from the application program may result in random background information from a previous EDF display appearing on the screen.
- An input command may be executed against the previous EDF display, rather than a display from the application program, or, if it is the first receive in the transaction, it may require explicit input from the CEDF panel instead of being satisfied by the contents of the initial ttoa.

These considerations apply to any screen I/O operation performed by the application program.

When EDF restores the transaction display, it does not sound the alarm or affect the keyboard in the same way as the user transaction. The effect of the user transaction options is seen when the SEND command is processed, but not when the screen is restored. When you have NOEDF specified in single-screen mode, you should take care that your program does not send and receive data because you will not see it.

When EDF restores the transaction display on a device that uses color, programmed symbols, or extended highlighting, these attributes are no longer present and the display is monochrome without the programmed symbols or extended highlighting. Also, if the inbound reply mode in the application program is set to "character" to enable the attribute-setting keys, EDF resets this mode, causing these keys to be disabled. If these changes prevent your transaction from executing properly, you should test in a dual-screen mode.

If you end your EDF session part way through the transaction, EDF restores the screen with the keyboard locked if the most recent RECEIVE command has not been followed by a SEND command; otherwise, the keyboard is unlocked.

Checking pseudoconversational programs

EDF makes a special provision for testing pseudoconversational transactions from a single terminal. If the terminal came out of EDF mode between the several tasks that make up a pseudoconversational transaction, it would be very hard to do any debugging after the first task. So, when a task terminates, EDF asks the operator whether EDF mode is to continue to the next task. If you are debugging a pseudoconversational task, press enter, as the default is "yes". If you have finished, reply "no".

Using EDF in dual-screen mode

In dual-screen mode, you use one terminal for EDF interaction and another for sending input to, and receiving output from, the transaction under test.

You start by entering, at the EDF terminal, the transaction CEDF tttt, where tttt is the name of the terminal on which the transaction is to be tested.

The message that CEDF gives in response to this depends on whether there is already a transaction running on the second terminal. If the second terminal is not busy, the message displayed at the first terminal is:

```
TERMINAL tttt: EDF MODE ON
```

and nothing further happens until a transaction is started on the second terminal, when the PROGRAM INITIATION display appears.

You can also use EDF in dual-screen mode to monitor a transaction that is already running on the second terminal. If, for example, you believe a transaction at a specific terminal to be looping, you can go to another terminal and enter a CEDF transaction naming the terminal at which this transaction is running. The message displayed at the first terminal is:

```
TERMINAL tttt: TRANSACTION RUNNING: EDF MODE ON
```

EDF picks up control at the next EXEC CICS command executed, and you can then observe the sequence of commands that are causing the loop, assuming that at least one EXEC CICS command is executed.

EDF and remote transactions

You cannot use EDF in dual-screen mode if the transaction under test, or the terminal that invokes it, is owned by another CICS region.

Furthermore, if the remote CICS region is earlier than CICS/ESA 3.1.1, you cannot run the transaction directly under EDF by invoking CEDF in the TOR. In this situation, you must use the routing transaction, CRTE. You enter CEDF at the terminal, clear the screen, and then enter CRTE followed by the system identifier (SYSIDNT) of the remote CICS region. This action causes CICS to route subsequent inputs to the remote region, and you can then enter the transaction identifier of the transaction you want to test. The *CICS Supplied Transactions* manual explains how to use CRTE.

If a remote transaction abends while under EDF using a CRTE routing session, EDF displays the abnormal task termination screen, followed by message DFHAC2206 for the user transaction. The CRTE session is not affected by the user task abend. Also, if you opted to continue with EDF after the abend, your terminal remains in EDF mode within the CRTE routing session.

There is a difference in execution as well. For remote transactions, EDF purges its memory of your session at the termination of each transaction, whether EDF is to be continued or not. This means that any options you have set and any saved screens are lost between the individual tasks in a pseudoconversational sequence.

EDF and non-terminal transactions

You can use EDF to test transactions that execute without a terminal: for example, transactions started by an EXEC CICS START command, or transactions initiated

by a transient data trigger-level. To test non-terminal transactions, use the CEDX *trnx* command, where *trnx* is the transaction identifier.

To test a transaction using CEDX:

- The terminal you use for the EDF displays, at which you enter the CEDX command, must be logged on to the CICS region in which the specified transaction is to execute.
- The CEDX command must be issued before the specified transaction is started by CICS. Other instances of the same transaction that are already executing when you issue the CEDX command are ignored.

When you use CEDX to debug a transaction, CICS controls the EDF operation by modifying the definition of the transaction specified on the CEDX command, to reference a special transaction class, DFHEDFTC. When you switch off EDF (using CEDX *tranid,OFF*) CICS modifies the transaction definition back to its normal transaction class.

EDF and DTP programs

You can also test a transaction that is using distributed transaction processing across a remote link by telling EDF to monitor the session on the link. You can do this on either (or both) of the participating systems that are running under CICS and has EDF installed. (You cannot do this if the transaction has been routed from another CICS region because you must use single-screen mode for remote transactions.)

For APPC and MRO links, you can name the system identifier (sysid) of the remote system:

```
CEDF sysid
```

This causes EDF to associate itself with any transaction attached across any session belonging to the specified system.

For APPC, MRO, and LU6.1 links, you can use the session identifier (sessionid) that the transaction is using:

```
CEDF sessionid
```

You can determine the session identifier with the CEMT INQUIRE TERMINAL transaction, but this means that the transaction must be running and have reached the point of establishing a session before you start EDF.

If a transaction using distributed transaction processing also has a terminal associated with it, or if you can invoke it from a terminal (even though it does not use one), you can use EDF to test it in the ordinary way from that terminal.

When you have finished testing the transaction on the remote system, you should turn off EDF on that SYSID or sessionid before logging off from CICS with CESF. For example:

```
CEDF sysid,OFF
```

Failure to do this could cause another transaction using a link to that system to be suspended.

EDF and distributed program link commands

You can use EDF, in single- or dual-terminal mode, to test a transaction that includes a distributed program link (DPL) command. However, EDF displays only

the DPL command invocation and response screens. CICS commands issued by the remote program are not displayed, but a remote abend, and the message **a remote abend has occurred** is returned to the EDF terminal, along with the SYSID of the system from which the abend was received. After control is returned to your local program, EDF continues to test as normal, but the PSW is not displayed if the abend is in a remote program.

Stopping EDF

If you want to end EDF control of a terminal, the method you use depends on where you are in the testing. If the transaction under test is still executing and you want it to continue, but without EDF, press the END EDF SESSION function key. If you have reached the task termination intercept, EDF asks if you want to continue. If you do not, overwrite the reply as NO (YES is the default). If no transaction is executing at the terminal, clear the screen and enter:

```
CEDF ,OFF
```

(The space and comma are required.)

If you are logging off from dual-screen mode, clear the screen and enter CEDF tttt,OFF.

In all these cases, the message **THIS TERMINAL: EDF MODE OFF** is displayed at the top of an empty screen.

Overtyping to make changes

Most of the changes you make with EDF involve changing information in memory. You do this simply by typing over the information shown on the screen with the information you want used instead. You can change any area where the cursor stops when you use the tab keys, except for the menu area at the bottom.

When you change the screen, you must observe the following rules:

- On CICS command screens, any argument value can be overtyped, but not the keyword of the argument. An optional argument cannot be removed, nor can an option be added or deleted.
- When you change an argument in the command display (as opposed to the working storage screen), you can change only the part shown on the display. If you attempt to overwrite beyond the value displayed, the changes are not made and no diagnostic message is generated. If the argument is so long that only part of it appears on the screen, you should change the area in working storage to which the argument points. (To determine the address, display the argument in hexadecimal format; the address of the argument location also appears.)
- In character format, numeric values always have a sign field, which can be overtyped with a minus or a blank only.
- When an argument is to be displayed in character format, some of the characters may not be displayable (including lowercase characters). EDF replaces each nondisplayable character with a period. When overtyping a period, you must be aware that the storage may in fact contain a character other than a period. You should not overwrite any character with a period; if you do, the change is ignored and no diagnostic message is issued. If you need to overwrite a character with a period, you can do so by switching the display to hexadecimal format, using PF2, and overtyping with X'4B'.

- When storage is displayed in both character and hexadecimal format and changes are made to both, the value of the hexadecimal field takes precedence should the changes conflict; no diagnostic message is issued.
- The arguments for some commands, such as HANDLE CONDITION, are program labels rather than numeric or character data. The form in which EDF displays (and accepts modifications to) these arguments depends on the programming language in use:
 - For COBOL, a null argument is displayed: for example, ERROR (), and because of this, you cannot modify it.
 - For C and C++, labels are not valid.
 - For PL/I, the address of the label constant is used; for example, ERROR (X'001D0016').
 - For assembler language, the address of the program label is used; for example, ERROR (X'00030C').

If no label value is specified on a HANDLE CONDITION command, EDF displays the condition name alone without the parentheses.

- The response field can be overtyped with the name of any exception condition, including ERROR, that can occur for the current function, or with the word NORMAL. The effect when EDF continues is that the program takes whatever action has been prescribed for the specified response. You can get the same effect by changing the EIBRESP field in the EIB display to the corresponding values. If you change the EIBRESP value or the response field on the **command execution complete** screen, EIBRCODE is updated. EIBRESP appears on second EIB screen and is the only one you can change (EIBRCODE protected). You can get the same effect by changing the EIBRESP value on the EIB display; EDF changes related values in the EIB and command screens accordingly if you do this.
- If uppercase translation is not specified for the terminal you are using you must take care to always enter uppercase characters.
- Any command can be overtyped with NOOP or NOP before processing; this suppresses processing of the command. Use of the ERASE EOF key, or overtyping with blanks, gives the same effect. When the screen is redisplayed with NOOP, the original verb line can be restored by erasing the whole verb line with the ERASE EOF key and pressing the ENTER key.

When you overtype a field representing a data area in your program, the change is made directly in application program storage and is permanent. However, if you change a field that represents a constant (a program literal), program storage is not changed, because this may affect other parts of the program that use the same constant or other tasks using the program. The command is executed with the changed data, but when the command is displayed after processing, the original argument values reappear. For example, suppose you are testing a program containing a command coded:

```
EXEC CICS SEND MAP('MENU') END-EXEC.
```

If you change the name MENU to MENU2 under EDF before executing the command, the map actually used is MENU2, but the map displayed on the response is MENU. (You can use the “previous display” key to verify the map name you used.) If you process the same command more than once, you must enter this type of change each time.

EDF responses

The response of EDF to any keyboard entry follows the rules listed below, in the order shown:

1. If the CLEAR key is used, EDF redisplay the screen with any changes ignored.
2. If invalid changes are made, EDF accepts any valid changes and redisplay the screen with a diagnostic message.
3. If the display number is changed, EDF accepts any other changes and shows the requested display.
4. If a PF key is used, EDF accepts any changes and performs the action requested by the PF key. Pressing ENTER with the cursor under a PF key definition in the menu at the bottom of the screen is the same as pressing a PF key.
5. If the ENTER key is pressed and the screen has been modified (other than the REPLY field), EDF redisplay the screen with changes included.
6. If the ENTER key is pressed and the screen has not been modified (other than the REPLY field), the effect differs according to the meaning of the ENTER key. If the ENTER key means CONTINUE, the user transaction continues to execute. If it means CURRENT DISPLAY, EDF redisplay the current status display.

Using EDF menu functions

The function keys that you can use at any given time are displayed in a menu at the bottom of every EDF display (see Figure 134 on page 611). The function of the ENTER key for that display is also shown. Functions that apply to all displays are always assigned to the same key, but definitions of some keys depend on the display and the intercept point. To select an option, press the indicated function key. Where a terminal has 24 function keys, EDF treats PF13 through PF24 as duplicates of PF1 through PF12 respectively. If your terminal has no PF keys, place the cursor under the option you want and press the ENTER key.

ABEND USER TASK

terminates the task being monitored. EDF asks you to confirm this action by displaying the message "ENTER ABEND CODE AND REQUEST ABEND AGAIN". After entering the code at the position indicated by the cursor, the user must request this function again to abend the task with a transaction dump identified by the specified code. If you enter "NO", the task is abended without a dump and with the 4-character default abend code of four question marks (????).

Abend codes beginning with the character A are reserved for use by CICS. Using a CICS abend code may cause unpredictable results.

You cannot use this function if an abend is already in progress or the task is terminating.

BROWSE TEMP STORAGE

produces a display of the temporary storage queue CEBRxxxx, where xxxx is the terminal identifier of the terminal running EDF. This function is only available from the working storage (PF5) screen. You can then use CEBR commands, discussed in "Using the CEBR commands" on page 637, to display or modify temporary storage queues and to read or write transient data queues.

CONTINUE

redisplay the current screen if you have made any changes, incorporating the

changes. If you had not made changes, CONTINUE causes the transaction under test to resume execution up to the next intercept point. To continue, press ENTER.

CURRENT DISPLAY

redispays the current screen if you have made any changes, with the changes incorporated. If you have not made changes, it causes EDF to display the command screen for the last intercept point. To execute this function, press ENTER from the appropriate screen.

DIB DISPLAY

shows the contents of the DL/I interface block (DIB). This function is only available from the working-storage screen (PF5). See the *Application Programming: EXEC DLI Commands* manual for information on DIB fields.

EIB DISPLAY

displays the contents of the EIB. See Figure 135 on page 612 for an example of an EIB display. For programming information about the EIB, see the *CICS Application Programming Reference* manual. If COMMAREA exists, EDF also displays its address and one line of data in the dump format.

INVOKE CECI

accesses CECI. This function is only available from the working storage (PF5) screen. See Figure 146 on page 620 for an example of the screen from which CECI is invoked. You can then use CECI commands, discussed in Chapter 52, “Command-level interpreter (CECI),” on page 641. These CECI commands include INQUIRE and SET commands against the resources referenced by the original command before and after command execution. See “inbound reply mode” on page 609 for restrictions when running CECI in dual-screen mode. The use of CECI from this panel is similar to the use of CEBR within CEDF.

END EDF SESSION

ends the EDF control of the transaction. The transaction continues running from that point but no longer runs in EDF mode.

NEXT DISPLAY

is the reverse of PREVIOUS DISPLAY. When you have returned to a previous display, this option causes the next one forward to be displayed and the display number to increase by one.

PREVIOUS DISPLAY

causes the previous display to be sent to the screen. This is the previous command display, unless you saved other displays. The number of the display from the current intercept point is always 00. As you request previous displays, the display number decreases by 1 to -01 for the first previous display, -02 for the one before that, and so on, down to the oldest display, -10. When no more previous screens are available, the PREVIOUS option disappears from the menu, and the corresponding function key becomes inoperative.

REGISTERS AT ABEND

displays storage containing the values of the registers should a local ASRA abend occur. The layout of the storage is:

- Register values (0 through 15)
- PSW at abend (8 bytes)

In some cases, when a second program check occurs in the region before EDF has captured the values of the registers, this function does not appear on the menu of the abend display. If this happens, a second test run generally proves to be more informative.

REMEMBER DISPLAY

places a display that would not usually be kept in memory, such as an EIB display, in the EDF memory. (EDF automatically saves the displays at the start and completion of each command.) The memory can hold up to 10 displays. The displays are numbered in reverse chronological order (that is, –10 is the oldest display, and –01 is the newest). All pages associated with the display are kept in memory and can be scrolled when recalled. Note, however, that if you save a working-storage display, only the screen on view is saved.

SCROLL BACK

applies to an EIB, DIB, or command display that does not all fit on one screen. When the screen on view is not the first one of the display, and there is a plus sign (+) before the first option or field, you can view previous screens in the display by selecting SCROLL BACK. See Figure 135 on page 612 for an example.

SCROLL FORWARD

applies to an EIB, DIB, or command display that does not all fit on one screen. When this happens, a plus sign (+) appears after the last option or field in the display, to show that there are more screens. Using SCROLL FORWARD brings up the next screen in the display.

SCROLL BACK FULL

has the same function for displays of working storage as the SCROLL BACK option for EIB and DIB displays. SCROLL BACK FULL gives a working-storage display one full screen backward, showing addresses lower in storage than those on the current screen.

SCROLL FORWARD FULL

has the same function for displays of working storage as the SCROLL FORWARD option for EIB and DIB displays. SCROLL FORWARD FULL gives a working-storage display one full screen forward, showing addresses higher in storage than those on the current screen.

SCROLL BACK HALF

is similar to SCROLL BACK FULL, except that the display of working storage is reversed by only half a screen.

SCROLL FORWARD HALF

is similar to SCROLL FORWARD FULL, except that the display of working storage is advanced by only half a screen.

STOP CONDITIONS

produces the menu screen shown in Figure 147 on page 629. You use this screen to tell EDF when to resume its displays after you have pressed the SUPPRESS DISPLAYS key. You can use STOP CONDITIONS and SUPPRESS DISPLAYS together to cut down on the interaction between you and EDF when you are checking a program that you know is partly working.

```
TRANSACTION: AC20 PROGRAM: DFH0VT1 TASK: 0086 APPLID: 1234567 DISPLAY: 00  
DISPLAY ON CONDITION:-
```

```
COMMAND:                                EXEC CICS  
OFFSET:                                X'.....'  
LINE NUMBER:  
CICS EXCEPTION CONDITION:              ERROR  
ANY CICS CONDITION                      NO  
TRANSACTION ABEND                      YES  
NORMAL TASK TERMINATION                YES  
ABNORMAL TASK TERMINATION              YES
```

```
DLI ERROR STATUS:  
ANY DLI ERROR STATUS
```

```
ENTER: CURRENT DISPLAY
```

```
PF1 : UNDEFINED      PF2 : UNDEFINED      PF3 : UNDEFINED  
PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE  PF6 : USER DISPLAY  
PF7 : UNDEFINED      PF8 : UNDEFINED      PF9 : UNDEFINED  
PF10: UNDEFINED      PF11: UNDEFINED     PF12: REMEMBER DISPLAY
```

Figure 147. Typical EDF display for STOP CONDITIONS

You can specify any or all of these events as STOP CONDITIONS:

- A specific type of function and option, such as READNEXT file or ENQ resource, is encountered, for example, FEPI ADD or GDS ASSIGN.
- The command at a specific offset or on a specific line number (assuming the program has been translated with the DEBUG option) is encountered.
- Any DLI error status occurs, or a particular DLI error status occurs.
- A specific exception condition occurs. If CICS exception condition is specified as ERROR (the default), EDF redisplay a screen in response to any ERROR condition (for example, NOTOPEN, EOF, or INVREQ). If you specify a specific condition such as EOF, EDF redisplay the screen only when an EOF condition arises, provided that ANY CICS CONDITION is left as the default NO.

If this field is changed to YES, EDF overrides the CICS exception conditions and redisplay a screen whenever any command results in a non-zero EIBRESP value such as NOTOPEN, EOF, or QBUSY.

- Any exception condition occurs for which the CICS action is to raise ERROR; for example, INVREQ or NOTFND.
- An abend occurs.
- The task ends normally.
- The task ends abnormally.

You do not always have to set STOP CONDITIONS in order to use the SUPPRESS DISPLAYS function, because EDF sets a default in the following fields on the assumption that you usually want to resume displays if any of them occurs:

- CICS exception condition
- Transaction abend
- Normal task termination
- Abnormal task termination

These are the options described in Figure 147 on page 629. You can turn off any of the defaults that do not apply when you bring up the STOP CONDITIONS menu, as well as adding conditions specific to your program.

When you use an offset for STOP CONDITIONS, you must specify the offset of the BALR instruction corresponding to a command. The offset can be determined from the code listing produced by the compiler or assembler. In COBOL, C, C++, or PL/I, you must use the compiler option that produces the assembler listing to determine the relevant BALR instruction.

When you use a line number, you must specify it exactly as it appears on the listing, including leading zeros, and it must be the line on which a command starts. If you have used the NUM or the SEQUENCE translator options, the translator uses your line numbers as they appear in the source. Otherwise, the translator assigns line numbers.

Line numbers can be found in the translator listing (SYSPRINT in the translator step) if you have used either the SOURCE or VBREF translator options. If you have used the DEBUG translator option, as you must to use line numbers for STOP CONDITIONS, the line number also appears in your compilation (assembly) listing, embedded in the translated form of the command, as a parameter in the CALL statement.

You can tell EDF to stop suppressing displays at DL/I commands as well as at CICS commands. You do this by overtyping the qualifier "CICS" on the command line with "DLI" and entering the type of DL/I command at which you want suppression to stop. You must be executing a DL/I program or have executed one earlier in the same task. You can suppress DL/I commands as early as the program initiation panel.

You can also stop suppression when a particular DL/I status code occurs. For information about the status codes that you can use, see the list of codes in the DL/I interface block (DIB) in the *Application Programming: EXEC DLI Commands* manual.

SUPPRESS DISPLAYS

suppresses all EDF displays until one of the specified STOP CONDITIONS occurs. When the condition occurs, however, you still have access to the 10 previous command displays, even though they were not actually sent to the screen when they were originally created.

SWITCH HEX/CHAR

switches displays between character and hexadecimal form. The switch applies only to the command display, and has no effect on previously remembered displays, STOP CONDITIONS displays, or working-storage displays.

In DL/I command displays which contain the WHERE option, only the key values (the expressions following each comparison operator) can be converted to hexadecimal.

UNDEFINED

means that the indicated function key is not defined for the current display at the current intercept point.

USER DISPLAY

causes EDF to display what would be on the screen if the transaction was not running in EDF mode. (You can use it only for single terminal checkout.) To return to EDF after using this key, press the ENTER key.

WORKING STORAGE

allows you to see the contents of the working-storage area in your program, or

of any other address in the CICS region. Figure 148 shows a typical working-storage screen.

```

TRANSACTION: AC20 PROGRAM: DFH0VT1 TASK: 00030 APPLID: 1234567 DISPLAY:00
ADDRESS: 035493F0 WORKING STORAGE
035493F0 000000 E3F14040 00000000 00010000 00000000 T1 .....
03549400 000010 00000000 00000000 F1000000 00000000 .....1.....
03549410 000020 F0000000 00000000 F0000000 00000000 0.....0.....
03549420 000030 F0000000 00000000 F0000000 00000000 0.....0.....
03549430 000040 00000000 00000000 00000000 00000000 .....
03549440 000050 D7C1D5D3 00000000 D9C5C3C4 00000000 PANL....RECD....
03549450 000060 D3C9E2E3 00000000 C8C5D3D7 00000000 LIST....HELP....
03549460 000070 84000000 00000000 A4000000 00000000 d.....u.....
03549470 000080 82000000 00000000 C4000000 00000000 b.....D.....
03549480 000090 E4000000 00000000 C2000000 00000000 U.....B.....
03549490 0000A0 D5000000 00000000 E2000000 00000000 N.....S.....
035494A0 0000B0 7B000000 00000000 6C000000 00000000 #.....%.....
035494B0 0000C0 4A000000 00000000 F1000000 00000000 ¢.....1.....
035494C0 0000D0 F2000000 00000000 F3000000 00000000 2.....3.....

ENTER: CURRENT DISPLAY
PF1 : UNDEFINED PF2 : BROWSE TEMP STORAGE PF3 : UNDEFINED
PF4 : EIB DISPLAY PF5 : INVOKE CECI PF6 : USER DISPLAY
PF7 : SCROLL BACK HALF PF8 : SCROLL FORWARD HALF PF9 : UNDEFINED
PF10: SCROLL BACK FULL PF11: SCROLL FORWARD FULL PF12: REMEMBER DISPLAY

```

Figure 148. Typical EDF display for working-storage

The working-storage contents are displayed in a form similar to that of a dump listing, that is, in both hexadecimal and character representation. The address of working storage is displayed at the top of the screen. You can browse through the entire area using the scroll commands, or you can simply enter a new address at the top of the screen. This address can be anywhere within the CICS region. The **working-storage** display provides two additional scrolling keys, and a key to display the EIB (the DIB if the command is a DL/I command).

The meaning of “working storage” depends on the programming language of the application program, as follows:

COBOL

All data storage defined in the WORKING-STORAGE section of the program

C, C++ and PL/I

The dynamic storage area (DSA) of the current procedure

Assembler language

The storage defined in the current DFHEISTG DSECT

Assembler language programs do not always acquire working storage; it may not be necessary, for example, if the program does not issue CICS commands. You may get the message “Register 13 does not address DFHEISTG” when you LINK to such a program. The message does not necessarily mean an error, but there is no working storage to look at.

Except for COBOL programs, working storage starts with a standard format save area; that is, registers 14 to 12 begin at offset 12 and register 13 is stored at offset 4.

Working storage can be changed at the screen; either the hexadecimal section or the character section can be used. Also, the ADDRESS field at the head of the

display can be overtyped with a hexadecimal address; storage starting at that address is then displayed when ENTER is pressed. This allows any location in the address space to be examined. Further information on the use of overtyping is given in “Overtyping to make changes” on page 624.

If the program storage examined is not part of the working storage of the program currently executing (which is unique to the particular transaction under test), the corresponding field on the screen is protected to prevent the user from overwriting storage that might belong to or affect another task.

If the initial part of a working-storage display line is blank, the blank portion is not part of working storage. This can occur because the display is doubleword aligned.

At the beginning and end of a task, working storage is not available. In these circumstances, EDF generates a blank storage display so that the user can still examine any storage area in the region by overtyping the address field.

Note that if you terminate a PL/I or Language Environment program with an ordinary non-CICS return, EDF does not intercept the return, and you are not able to see working storage. If you use a RETURN command instead, you get an EDF display before execution and at program termination.

If you are using a Language Environment-enabled program, working storage is freed at program termination if the program is terminated using a non-CICS return. In this case, working storage is not available for display.

Chapter 51. Temporary storage browse (CEBR)

You can use the browse transaction (CEBR) to browse temporary storage queues and delete them. You can also use the CEBR transaction to transfer the contents of a transient data queue to temporary storage in order to look at them, and to reestablish the transient data queue when you have finished. The CEBR commands that perform these transfers allow you to add records to a transient data queue and remove all records from a transient data queue.

Some installations restrict the use of the CEBR transaction, particularly in production systems, to prevent modifications that were not intended or not authorized. Installations also may protect individual resources, including temporary storage and transient data queues. If you are using the CEBR transaction and experience an abend described as a security failure, you probably have attempted to access a queue to which your user ID is not authorized.

This chapter describes:

- “Using the CEBR transaction”
- “What does the CEBR transaction display?” on page 635
- “Using the CEBR function keys” on page 636
- “Using the CEBR commands” on page 637
- “Using the CEBR transaction with transient data” on page 639

Using the CEBR transaction

You start the CEBR transaction by entering the transaction identifier CEBR, followed by the name of the queue you want to browse. You can choose a name of up to 16 characters. For example, to display the temporary storage queue named AXBYQUEUEUENAME111 you type CEBR AXBYQUEUEUENAME111 and press ENTER. If the queue name includes lower case characters, ensure that upper case translation is suppressed for the terminal you are using, and then enter the correct combination of upper and lower case characters. CICS responds with a display of the queue, for example, as shown in Figure 149 on page 634.

Alternatively, you can start the CEBR transaction from the CEDF transaction. You do this by pressing PF5 from the initial CEDF screen (see Figure 134 on page 611) which takes you to the working-storage screen, and then pressing PF2 from that screen to browse temporary storage (that is, invoke the CEBR transaction). CEBR can also be started from CEMT I TSQ by entering 'b' at the queue to be browsed. The CEBR transaction responds by displaying the temporary storage queue whose name consists of the four letters CEBR followed by the four letters of your terminal identifier. (CICS uses this same default queue name if you invoke the CEBR transaction directly and do not supply a queue name.) The result of invoking the CEBR transaction without a queue name or from an EDF session at terminal S21A is shown in Figure 150. If you enter the CEBR transaction from the CEDF transaction, you return to the EDF panel when you press PF3 from the CEBR screen.

```

CEBR TSQ AXBYQUEUEAME111 SYSID CIJP REC 1 OF 3 COL 1 OF 5
ENTER COMMAND ==>
***** TOP OF QUEUE *****
00001 HELLO
00002 HELLO
00003 HELLO
***** BOTTOM OF QUEUE *****

PF1 : HELP          PF2 : SWITCH HEX/CHAR    PF3 : TERMINATE BROWSE
PF4 : VIEW TOP      PF5 : VIEW BOTTOM      PF6 : REPEAT LAST FIND
PF7 : SCROLL BACK HALF PF8 : SCROLL FORWARD HALF PF9 : UNDEFINED
PF10: SCROLL BACK FULL PF11: SCROLL FORWARD FULL PF12: UNDEFINED

```

Figure 149. Typical CEBR display of temporary storage queue contents

```

CEBR TSQ AXBYQUEUEAME1AA SYSID CIJP REC 1 OF 0 COL 1 OF 1
ENTER COMMAND ==>
***** TOP OF QUEUE *****
***** BOTTOM OF QUEUE *****

TS QUEUE AXBYQUEUEAME1AA DOES NOT EXIST
PF1 : HELP          PF2 : SWITCH HEX/CHAR    PF3 : TERMINATE BROWSE
PF4 : VIEW TOP      PF5 : VIEW BOTTOM      PF6 : REPEAT LAST FIND
PF7 : SCROLL BACK HALF PF8 : SCROLL FORWARD HALF PF9 : UNDEFINED
PF10: SCROLL BACK FULL PF11: SCROLL FORWARD FULL PF12: UNDEFINED

```

Note: 1 Header 2 Command area 3 Body 4 Message line 5 Menu of options

Figure 150. Typical CEBR display of default temporary storage queue

What does the CEBR transaction display?

As shown in Figure 150 on page 634, a CEBR transaction display consists of a header, a command area, a body (the primary display area), a message line, and a menu of functions you can select at this point.

The header

The header shows:

- The transaction being run, that is, CEBR.
- The identifier of the temporary storage queue (AXBYQUEUEAME111 in Figure 149 on page 634 and (AXBYQUEUEAME1AA in Figure 150 on page 634). You can overwrite this field in the header if you want to switch the screen to another queue. If the queue name includes lower case characters, ensure that upper case translation is suppressed for the terminal you are using, and then enter the correct combination of upper and lower case characters.
- The system name that corresponds to a temporary storage pool name or to a remote system. If you have not specified one, the name of the local system is displayed. You can overwrite this field in the header if you want to browse a shared or remote queue.
- The number of the highlighted record.
- The number of records in the queue (three in AXBYQUEUEAME111 and none in AXBYQUEUEAME1AA)
- The position in each record at which the screen starts (position 1 in both cases) and the length of the longest record (22 for queue AXBYQUEUEAME111 and zero for queue AXBYQUEUEAME1AA).

The command area

The command area is where you enter commands that control what is to be displayed and what function is to be performed. These commands are described in “Using the CEBR commands” on page 637. You can also modify the screen with function keys shown in the menu of options at the bottom of the screen. The function keys are explained in “Using the CEBR function keys” on page 636.

The body

The body is where the queue records are shown. Each line of the screen corresponds to one queue record. If a record is too long for the line, it is truncated. You can change the portion of the record that is displayed, however, so that you can see an entire record on successive screens. If the queue contains more records than will fit on the screen, you can page forward and backward through them, or specify at what record to start the display, so that you can see all the records you want.

The message line

CEBR uses the message line between the body and menu to display messages to the user, such as the “Does not exist” message in Figure 150 on page 634.

Using the CEBR function keys

The function keys that you can use at any time are displayed at the bottom of every CEBR transaction screen. The keys have the same meaning on all screens. If your terminal does not have PF keys, you can simulate their use by placing the cursor under the description and pressing ENTER. Where a terminal has 24 function keys, the CEBR transaction treats PF13 through PF24 as duplicates of PF1 through PF12 respectively.

PF1 HELP

Displays a help screen that lists all the commands you can use when the CEBR transaction is running. You can return to the main screen by pressing ENTER.

PF2 SWITCH HEX/CHAR

Switches the screen from character to hexadecimal format, and back again.

PF3 TERMINATE BROWSE

Terminates the CEBR transaction. If you entered the CEBR transaction directly, it frees up your terminal for the next transaction. If you entered from an EDF session, it returns you to the working-storage screen from which you entered. If you entered from CEMT I TSQ, it returns you to the CEMT screen.

PF4 VIEW TOP

Displays the first records in the queue and has the same effect as the TOP command.

PF5 VIEW BOTTOM

Displays the last records in the queue and has the same effect as the BOTTOM command.

PF6 REPEAT LAST FIND

Repeats the previous FIND command.

PF7 SCROLL BACK HALF

Moves the display backward by one-half the number of records that fit on the screen, so that the records on the top half of the screen move to the bottom half.

PF8 SCROLL FORWARD HALF

Advances the display by one-half the number of records that fit on the screen, so that the records on the bottom half of the screen move to the top half.

PF9 VIEW RIGHT (or VIEW LEFT)

Changes the screen to show the columns immediately after (to the right of) or before (to the left of) the columns currently on display. The key is not defined if the entire record fits on one line of the screen. It moves you to the right until the end of the record is reached, and then reverses to move left back to the beginning of the record. You can also use the COLUMN command to change the column at which the display begins.

PF10 SCROLL BACK FULL

Moves the screen backward by the number of records that fit on the screen, to show the records immediately before those currently on display.

PF11 SCROLL FORWARD FULL

Advances the screen by the number of records that will fit on the screen, to show the records immediately after those currently on display.

Using the CEBR commands

Here is a list of the CEBR commands that you can use to view and manipulate the records in the temporary storage queue.

BOTTOM

(Abbreviation: B)

Shows the last records in the temporary storage queue (as many as fill up the body of the screen, with the last record on the last line).

COLUMN nnnn

(Abbreviation: C nnnn)

Displays the records starting at character position (column) nnnn of each record. The default starting position, assumed when you initiate the CEBR transaction, is the first character in the record.

FIND /string

(Abbreviation: F /string)

Finds the next occurrence of the specified string. The search starts in the record *after* the **current record**. The current record is the one that is highlighted. In the initial display of a queue, the current record is set to one, and therefore the search begins at record two.

If the string is found, the record containing the string becomes the highlighted line, and the display is changed to show this record on the second line. If you cannot see the search string after a successful FIND, it is in columns of the record beyond those on display; use the scroll key or the COLUMN command to shift the display right or left to show the string.

For example:

```
FIND /05-02-93
```

locates the next occurrence of the string "05-02-93" The / character is a delimiter. It does not have to be /, but it must not be a character that appears in the search argument. For example, if the string you were looking for was "05/02/93" instead of "05-02-93", you could not use the following:

```
FIND /05/02/93
```

There is a slash in the search string. The following examples would work:

```
FIND X05/02/93      or      FIND S05/07/93
```

Any delimiter except a / or one of the digits in the string works. If there are any spaces in the search string, you must repeat the delimiter at the end of the string. For example:

```
FIND /CLARE JACKSON/
```

The search string is not case-sensitive. When you have entered a FIND command, you can repeat it (that is, find the next occurrence of the string) by pressing PF6.

GET xxxx

(Abbreviation: G xxxx)

Transfers the named transient data queue to the end of the temporary storage queue currently on display. This enables you to browse the contents of the queue. xxxx must be either the name of an intrapartition transient data queue, or the name of an extrapartition transient data queue that has been opened for input. See "Using the CEBR transaction with transient data" on page 639 for more information about browsing transient data queues.

LINE nnnn**(Abbreviation: L nnnn)**

Starts the body of the screen at the queue record one prior to nnnn, and sets the current line to nnnn. (This arrangement causes a subsequent FIND command to start the search after record nnnn.)

PURGE

Deletes the queue being browsed.

Do not use PURGE to delete the contents of an internally generated queue, such as a BMS logical message.

Note: If you purge a recoverable temporary storage queue, no other task can update that queue (add a record, change a record, or purge) until your task ends.

PUT xxxx**(Abbreviation: P xxxx)**

Copies the temporary storage queue that is being browsed to the named transient data queue. xxxx must be either the name of an intrapartition transient data queue, or the name of an extrapartition transient data queue that has been opened for output. See “Using the CEBR transaction with transient data” on page 639 for more information about creating or restoring a transient data queue.

QUEUE xxxxxxxxxxxxxxxx**(Abbreviation: Q xxxxxxxx)**

Changes the name of the queue you are browsing. The value that you specify can be in character format using up to 16 characters (for example, QUEUE ABCDEFGHIJKLMNOP) or in hexadecimal format (for example, QUEUE X'C1C2C3C4'). If the queue name includes lower case characters, ensure that upper case translation is suppressed for the terminal you are using, and then enter the correct combination of upper and lower case characters. The CEBR transaction responds by displaying the data that is in the named queue.

You can also change the queue name by overtyping the current value in the header.

SYSID xxxx**(Abbreviation: S xxxx)**

Changes the name of the temporary storage pool or remote system where the queue is to be found.

You can also change this name by overtyping the current SYSID value in the header.

Note: If ISC is not active in the CICS system on which the CEBR transaction is running then the SYSID will default to the local SYSID.

TERMINAL xxxx**(Abbreviation: TERM xxxx)**

Changes the name of the queue you are browsing, but is tailored to applications that use the convention of naming temporary storage queues that are associated with a terminal by a constant in the first four characters and the terminal name in the last four. The new queue name is formed from the first four characters of the current queue name, followed by xxxx.

TOP

(Abbreviation: T)

Causes the CEBR transaction to start the display at the first record in the queue.

Using the CEBR transaction with transient data

The GET command reads each record in the transient data queue that you specify and writes it at the end of the temporary storage queue you are browsing, until the transient data queue is empty. You can then view the records that were in the transient data queue. When you have finished your inspection, you can copy the temporary storage queue back to the transient data queue (using the PUT command). This usually leaves the transient data queue as you found it, but not always. Here are some points you need to be aware of when using the GET and PUT commands:

- If you want to restore the transient data queue unchanged after you have browsed it, make sure that the temporary storage queue on display at the time of the GET command is empty. Otherwise, the existing temporary storage records is copied to the transient data queue when the subsequent PUT command is issued.
- After you get a transient data queue and before you put it back, other tasks may write to that transient data queue. When you issue your PUT command, the records in the temporary storage queue are copied **after** the new records, so that the records in the queue are no longer in the order in which they were originally created. Some applications depend on sequential processing of the records in a queue.
- After you get a **recoverable** transient data queue, no other task can access that queue until your transaction ends. If you entered the CEBR transaction from the CEDF transaction, the CEDF transaction must end, although you can respond “yes” to the “continue” question if you are debugging a pseudoconversational sequence of transactions. If you invoked the CEBR transaction directly, you must end it.
- Likewise, after you issue a PUT command to a recoverable transient data queue, no other task can access that queue until your transaction ends.

The GET and PUT commands do not need to be used as a pair. You can add to a transient data queue from a temporary storage queue with a PUT command at any time. If you are debugging code that reads a transient data queue, you can create a queue in temporary storage (with the CECI transaction, or the CEBR GET command, or by program) and then refresh the transient data queue as many times as you like from temporary storage. Similarly, you can empty a transient data queue by using a GET command without a corresponding PUT command.

Chapter 52. Command-level interpreter (CECI)

You can use the command-level interpreter (CECI) transaction to check the syntax of CICS commands and process these commands interactively on a 3270 screen. CECI allows you to follow through most of the commands to execution and display the results. It also provides you with a reference to the syntax of the whole of the CICS command-level application programming and system programming interface.

CECI interacts with your test system to allow you to create or delete test data, temporary storage queues, or to deliberately introduce wrong data to test out error logic. You can also use CECI to repair corrupted database records on your production system.

The interpreter is such a powerful tool that your installation may restrict its use with attach-time security. (The external security manager used by your installation defines the security attributes for the CECI and CECS transactions.) If this has been done, and you are not authorized to use the interpreter transaction you select, you will not be able to initiate the transaction.

This chapter describes:

- “What does CECI display?”
- “Using CECI” on page 648
- “Using the CECI function keys” on page 650
- “Saving commands” on page 653
- “How CECI runs” on page 654

What does CECI display?

All CECI screens have the same basic layout. As shown in Figure 153 on page 648, CECI displays consist of:

- “The command line”
- “The status line” on page 642
- “The body” on page 646
- “The message line” on page 646
- “CECI options on function keys” on page 646

The command line

The command line is the first line of the screen. You enter the command you want to process or whose syntax you want to check here. This can be the full or abbreviated syntax. The rules for entering and abbreviating the command are:

- The keywords EXEC CICS are optional.
- The options of a command can be abbreviated to the number of characters sufficient to make them unique. Valid abbreviations are shown in uppercase characters in syntax displays in the body of the screen.
- The quotes around character strings are optional, and all strings of characters are treated as character-string constants unless they are preceded by an ampersand (&), in which case they are treated as variables.
- Options of a command that receive a value from CICS when the command is processed are called **receivers**, and need not be specified. The value received from CICS is included in the syntax display, and stored in the variable if one has been specified, after the command has been processed.

- If you issue a CECI command with two of the keywords in conflict, CECI ignores the first keyword and issues an error message, such as this one, from a READ command:
E INTO option conflicts with SET option and is ignored
- If you put a question mark in front of your command, the interpreter stops after the syntax check, even if you have used the transaction code CECI. If you want to proceed with execution, remove the question mark.

The following example shows the abbreviated form of a command. The file control command:

```
EXEC CICS READ FILE('FILEA') RIDFLD('009000') INTO(&REC)
```

can be entered on the command input line, as:

```
READ FIL(FILEA) RID(009000)
```

or at a minimum, as:

```
READ F(FILEA) RI(009000)
```

In the first form, the INTO specification creates a variable, &REC, into which the data is to be read. However, INTO is a receiver (as defined above) and you can omit it. When you do, CICS creates a variable for you automatically.

The status line

As you go through the process of interpreting a command, CECI presents a sequence of displays. The format of the body of the screen is essentially the same for all; it shows the syntax of the command and the option values selected. The status line on these screens tells you where you are in the processing of the command, and is one of:

- COMMAND SYNTAX CHECK
- ABOUT TO EXECUTE COMMAND
- COMMAND EXECUTION COMPLETE
- COMMAND NOT EXECUTED

From any of these screens, you can select additional displays. When you do, the body of the screen shows the information requested, and the status line identifies the display, which may be any of:

- EXPANDED AREA
- VARIABLES
- EXEC INTERFACE BLOCK
- SYNTAX MESSAGES

These screens are described in “Using the CECI function keys” on page 650. You can request them at any time during processing and then return to the command interpretation sequence.

There is also one input field in the status line called NAME=. This field is used to create and name variables, as explained in “Variables” on page 650 and “Saving commands” on page 653.

Command syntax check

When the status line shows **command syntax check** (as shown in Figure 153 on page 648), it indicates that the command entered on the command input line has been syntax checked but is not about to be processed. This is always the status if you enter CECS or if you precede your command with a question mark. It is also the status when the syntax check of the command gives severe error messages.

In addition, you get this status if you attempt to execute one of the commands that the interpreter cannot execute. Although any command can be syntax-checked, using either CECS or CECI, the interpreter cannot process the following commands any further:

- EXEC CICS commands that depend upon an environment that the interpreter does not provide:
 - FREE
 - FREEMAIN
 - GETMAIN
 - HANDLE ABEND
 - HANDLE AID
 - HANDLE CONDITION
 - IGNORE CONDITION
 - POP HANDLE
 - PUSH HANDLE
 - SEND LAST
 - SEND PARTNSET
 - WAITCICS
 - WAIT EVENT
 - WAIT EXTERNAL
- BMS commands that refer to partitions (because the display cannot be restored after the screen is partitioned)
- EXEC DLI
- CPI Communication (CPI-C) commands
- SAA Resource Recovery interface (CPI-RR) commands

About to execute command

This display (as shown in Figure 151) appears when none of the reasons for stopping at **command syntax check** applies.

```
READ FILE('FILEA') RIDFLD('009000')
STATUS: ABOUT TO EXECUTE COMMAND NAME=
EXEC CICS READ
  File( 'FILEA  ' )
  < SYsid() >
  SEt() | Into()
  < Length() >
  RIdfld( '009000' )
  < Keylength() < GEneric > >
  < RBa | RRn | DEBRec | DEBKey >
  < GTeq | Equal >
  < Update < Token() > >
```

PF 1 HELP 2 HEX 3 END 4 EIB 5 VAR 6 USER 7 SBH 8 SFH 9 MSG 10 SB 11 SF

Figure 151. Typical CECI display for about to execute command

If you press the ENTER key at this point without changing the screen, CECI executes the command. You can still modify it at this point, however. If you do, CECI ignores the previous command and processes the new one from scratch. This means that the next screen displayed is **command syntax check** if the command cannot be executed or else **about to execute command** if the command is correct.

Command execution complete

This display (as shown in Figure 152) appears after the interpreter has executed a command, in response to the ENTER key from an unmodified **about to execute command** screen.

```
INQUIRE FILE NEXT
STATUS:  COMMAND EXECUTION COMPLETE                      NAME=
EXEC CICS INquire File( 'DFHCSD ' )
  < STArt | END | Next >
  < ACcessmethod( +0000000003 ) >
  < ADd( +00000000041 ) >
  < BASedsname( ' ' ) >
  < BLOCKFormat( +0000000016 ) >
  < BLOCKKeylen( -0000000001 ) >
  < BLOCKSize( -0000000001 ) >
  < BRowse( +0000000039 ) >
  < DElete( +0000000043 ) >
  < DIsposition( +0000000027 ) >
  < DSname( 'CFV01.CICS03.PSK.CSD ' ) >
  < EMptystatus( +0000000032 ) >
  < ENablestatus( +0000000033 ) >
  < EXclusive( +0000000001 ) >
  < Fwdrecstatus( +0000000361 ) >
  < Journalnum( +00000 ) >
+ < KEYLength( +0000000000 ) >

RESPONSE: NORMAL                      EIBRESP=+0000000000 EIBRESP2=+0000000000
PF 1 HELP 2 HEX 3 END 4 EIB 5 VAR 6 USER 7 SBH 8 SFH 9 MSG 10 SB 11 SF
```

Figure 152. Typical CECI display for command execution complete

The command has been processed and the results are displayed on the screen.

Any **receivers**, whether specified or not, together with their CICS-supplied values, are displayed intensified.

The body

The body of **command syntax check**, **about to execute command**, and **command execution complete** screens contains information common to all three displays.

The full syntax of the command is displayed. Options specified in the command line or assumed by default are intensified, to show that they are used in executing the command, as are any **receivers**. The < > brackets indicate that you can select an option from within these brackets. If you make an error in your syntax, CECI diagnoses it in the message area that follows the body, described in “The message line.” If there are too many diagnostic messages, the rest of the messages can be displayed using PF9.

Arguments can be displayed in either character or hexadecimal format. You can use PF2 to switch between formats. In character format, some characters are not displayable (including lowercase characters on some terminals); CECI shows them as periods. You need to switch to hexadecimal to show the real values, and you need to use caution when modifying them, as explained in “Making changes” on page 649.

If the value of an option is too long for the line, only the first part is displayed followed by “...” to indicate there is more. You can display the full value by positioning the cursor at the start of the option value and pressing ENTER. This produces an expanded display described in “Expanded area” on page 650.

If the command has more options than can fit on one screen, a plus sign (+) appears at the left-hand side of the last option of the current display to indicate that there are more. An example of this is shown in Figure 152 on page 645. You can display additional pages by scrolling with the PF keys.

The message line

CECI uses the message line to display error messages. After execution of a command, the message line shows the response code. Figure 153 on page 648 shows an error message, where the user has omitted a required field. The **S** that precedes the message indicates that it is severe (bad enough to prevent execution). There are also warning messages (flagged by **W**) and error messages (flagged by **E**), which provide information without preventing execution. **E** messages indicate option combinations unusual enough that they may not be intended and warrant a review of the command before you proceed with execution.

Where there are multiple error messages, CECI creates a separate display containing all of them, and uses the message line to tell you how many there are, and of what severity. You can get the message display with PF9, as explained in “Using the CECI function keys” on page 650.

Figure 152 on page 645 shows the second use of the message line, to show the result of executing a command. CECI provides the information in both text (NORMAL in the example in Figure 152 on page 645) and in decimal form (the EIBRESP and EIBRESP2 value).

CECI options on function keys

The single line at the foot of the screen provides a menu indicating the effect of the PF keys for the display.

The PF keys are described below. If the terminal has no PF keys, the same effect can be obtained by positioning the cursor under the required item in the menu and pressing ENTER.

PF1 HELP

displays a HELP panel giving more information on how to use the command interpreter and on the meanings of the PF keys.

PF2 HEX

(SWITCH HEX/CHAR) switches the display between hexadecimal and character format. This is a mode switch; all subsequent screens stay in the chosen mode until the next time this key is pressed.

PF3 END

(END SESSION) ends the current session of the interpreter.

PF4 EIB

(EIB DISPLAY) shows the contents of the EXEC interface block (EIB). An example of this screen is shown in Figure 155 on page 652.

PF5 VAR

(VARIABLES) shows all the variables associated with the current command interpreter session, giving the name, length, and value of each. See Variables for more information about the use of this PF key.

PF6 USER

(USER DISPLAY) shows the current contents of the user display panel (that is, what would appear on the terminal if the commands processed thus far had been executed by an ordinary program rather than the interpreter). This key is not meaningful until a terminal command is executed, such as SEND MAP.

PF7 SBH

(SCROLL BACK HALF) scrolls the body half a screen backward.

PF8 SFH

(SCROLL FORWARD HALF) scrolls the body half a screen forward.

PF9 MSG

(DISPLAY MESSAGES) shows all the messages generated during the syntax check of a command.

PF10 SB

(SCROLL BACK) scrolls the body one full screen backward.

PF11 SF

(SCROLL FORWARD) scrolls the body one full screen forward.

Using CECI

You start the command-level interpreter by entering either of two transaction identifiers, CECS or CECI, followed by the name of the command you want to test. You can list command options too, although you can also do this later. For example:

```
CECS READ FILE('FILEA')
```

or

```
CECI READ FILE('FILEA')
```

CICS responds with a display of the command and its associated functions, options, and arguments, as shown in Figure 153. If you leave out the command, CECI provides a list of possible commands to get you started. You can use any of the commands described for programming purposes in the *CICS Application Programming Reference* and the *CICS System Programming Reference* manuals. CECI also supports the FEPI commands provided for the CICS Front End Programming Interface.

```
READ FILE('FILEA')  
STATUS:  COMMAND SYNTAX CHECK  
EXEC CICS READ  
  File( 'FILEA  ' )  
  < SYsid() >  
  SEt() | Into()  
  < Length() >  
  RIDfld()  
  < Keylength() < GGeneric > >  
  < RBa | RRn | DEBRec | DEBKey >  
  < GTeq | Equal >  
  < Update < Token() > >  
  
S Option RIDFLD has been omitted or specified with an invalid value,  
  the command cannot be executed.  
PF 1 HELP 2 HEX 3 END 4 EIB 5 VAR 6 USER 7 SBH 8 SFH 9 MSG 10 SB 11 SF
```

Note: **1** Command line **2** Status line **3** Body **4** Message line **5** Menu of functions

Figure 153. Typical CECI display for command syntax check

If you use the transaction code CECS, the interpreter simply checks your command for correct syntax. If you use CECI, you have the option of executing your command once the syntax is correct. (CICS uses two transaction identifiers to allow different security to be assigned to syntax checking and execution.)

Making changes

Until CICS executes a command, you can change it by changing the contents of the command line, by changing the option values shown in the syntax display in the body, or by changing the values of variables on the **variables** screen. (You can still make changes after a command is executed, but, unless they are in preparation for another command, they have no effect.)

When you make your changes in the command line or on the **variables** screen, they last for the duration of the CECI transaction. If you make them in the body of the syntax screen, however, they are temporary. They last only until the command is executed and are not reflected in the command line.

As noted earlier, not all characters are displayable on all terminals. When the display is in character rather than hexadecimal format, CECI shows these characters as periods (X'4B'). When you overwrite a period, you should be aware that the current value may not be a period but an undisplayable character.

Furthermore, you cannot change a character to a period when the display is in character mode. If you attempt this, CECI ignores your change, and does not issue a diagnostic message. To make such a change, you have to switch the display to hexadecimal and enter the value (X'4B') that represents a period.

There is a restriction on changes in hexadecimal format as well. If you need to change a character to a blank, you cannot enter the code (X'40') from a hexadecimal display. Again, your change is ignored and CECI does not issue a message. Instead, you must switch to character mode and blank out the character.

After every modification, CECI rechecks your syntax to ensure that no errors have appeared. It restarts processing at the **command syntax check** if there are any execution-stoppers, and at **about to execute command** if not. Only after you press ENTER on an unmodified **about to execute command** screen does CECI execute your command.

Using the CECI function keys

Additional screens of information are available when you press the relevant PF key, described in “CECI options on function keys” on page 646. You can get back to your original screen by pressing ENTER from an unmodified screen.

Expanded area

This display uses the whole of the body of the screen to display a variable selected with the cursor. The cursor can be positioned at the start of the value of an option on a syntax display, or under the ampersand of a variable in a variables display. Pressing ENTER then gives the expanded area display. The scrolling keys can be used to display all the information if it exceeds a full screen.

Variables

Figure 154 shows the result of requesting a **variables** display, obtained by pressing PF5. For each variable associated with the current interpreter session, it shows the name, length, and value.

```
READ FILE('FILEA') RIDFLD('009000') INTO(&REC)
VARIABLES  LENGTH  DATA
&DFHC      +00016   THIS IS A SAMPLE
&DFHW      +00046   EXEC CICS WRITEQ QUEUE('  CIS200') FROM(&DFHC)
&DFHR      +00045   EXEC CICS READQ QUEUE('  CIS200') INTO(&DFHC)
&REC       +00080   482554 D694 72 WIDGET, .007 TEST 100

PF 1 HELP 2 HEX 3 END 4 EIB 5 VAR 6 USER          9 MSG
```

Figure 154. Typical CECI display of variables associated with a CECI session

The first three variables displayed are created for you by CECI and always appear unless you explicitly delete them. They are designed to help you create command lists, as described in “Saving commands” on page 653, as well as to serve as examples.

After these three, you see any variables that you have created. The fourth one in Figure 154, &REC, is the result of executing:

```
READ FILE('FILEA') RID('009000') INTO(&REC)
```

Normally, the value supplied for an option in the command line is taken as a character string constant. However, sometimes you need to specify a variable to represent this value, as when you want to connect two commands through option values.

For example, to change a record with CECI, you might first enter:

```
EXEC CICS READ UPDATE INTO(&REC)
      FILE('FILEA') RID('009000')
```

You would then modify the record as required by changing the variable &REC, and then enter:

```
EXEC CICS REWRITE FROM(&REC) FILE('FILEA')
```

The ampersand (&) in the first position tells CECI that you are specifying a variable.

A variable is also useful when the values of the options cause the command to exceed the line length of the command input area. Creating variables with the required values and specifying the variable names in the command overcomes the line length limitation.

Defining variables

Variables can have a data type of character, fullword, halfword, or packed decimal, and you can create them in any of the following ways:

- By naming the variable in a receiver (&REC in Figure 154 on page 650, for example). The variable is created when the command is processed. The data type and length are implied by the option.
- By adding new entries to the list of variables already defined. To create a new variable, simply type its name and length in the appropriate columns on the first unused line of the variables display, and then press ENTER. For character variables, use the length with which the variable has been defined. For fullwords or halfwords, type **F** or **H**. For packed variables, use the length in bytes, preceded by a **P**.

Character variables are initialized to blanks. The others are initialized to zero in the appropriate form. Once a variable is created, you can change the value by modifying the data field on the **variables** display.

- By using the NAME field on the status line when you have produced an expanded area display of a particular option. You do this by positioning the cursor under the option on a syntax display and pressing ENTER. Then you assign the variable name you want associated with the displayed option value by typing it into the NAME field and pressing ENTER again.
- By copying an existing variable. You do this by obtaining an expanded area display of the variable to be copied, overkeying the name displayed with the name of the new variable, and pressing ENTER.
- By using the NAME field directly on a syntax display. This creates a character variable whose contents are the character string on the command line, for use in command lists as explained in “Saving commands” on page 653.

You can also delete a variable, although you do not usually need to, as CECI discards all variables at session end. To delete one before session end, position the cursor under the ampersand that starts the name, press ERASE EOF, and then press ENTER.

The EXEC interface block (EIB)

You can display the EIB associated with the CECI transaction by pressing PF4. Figure 155 shows an example of the contents of the EXEC interface block (EIB).

```
READ FILE('FILEA') RIDFLD('009000')
EXEC INTERFACE BLOCK
  EIBTIME      = +0124613
  EIBDATE      = +0091175
  EIBTRNID     = 'CECI'
  EIBTASKN     = +0000046
  EIBTRMID     = 'S200'
  EIBCPOSN     = +00004
  EIBCALEN     = +00000
  EIBRID       = X'7D'
  EIBFN        = X'0000'
  EIBRCODE     = X'000000000000'
  EIBDS        = '.....'
  EIBREQID     = '.....'
  EIBSRCE      = ' '
  EIBSYNC      = X'00'
  EIBFREE      = X'00'
  EIBRECV      = X'00'
  EIBATT       = X'00'
  EIBEOC       = X'00'
+ EIBFMH      = X'00'

PF 1 HELP 2 HEX 3 END 4 EIB 5 VAR 6 USER 7 SBH 8 SFH 9 MSG 10 SB 11 SF
```

Figure 155. Typical CECI display of the EIB

The fields in the EIB are described for programming purposes in the *CICS Application Programming Reference* manual.

Error messages display

When there are more messages than CECI can display on the message line, you can display all of them by pressing PF9.

```
READ
SYNTAX MESSAGES
S FILE must be specified.
S RIDFLD must be specified.
```

```
PF 1 HELP 2 HEX 3 END 4 EIB 5 VAR 6 USER 7 SBH 8 SFH      10 SB 11 SF
```

Figure 156. Typical CECI display of the message display

Saving commands

Sometimes you may want to execute a command, or a series of commands, under CECI, repeatedly. One technique for doing this is to create a temporary storage queue containing the commands. You then alternate reading the command from the queue and executing it.

CECI provides shortcuts both for creating the queue and for executing commands from it. To create the queue:

1. Start a CECI session.
2. Enter the first (or next) command you want to save on the command line, put &DFHC in the NAME field in the status line, and press ENTER. This action causes the usual syntax check, and it also stores your command as the value of &DFHC, which is the first of those three variables that CECI always defines for you. (See Figure 154 on page 650.) If you select the variables display at this point, you will see that &DFHC is the value of your command.
3. After the syntax is correct but before execution (on the **about to execute command** screen), change the command line to &DFHW and press ENTER. This causes CECI to use the value of &DFHW for the command to be executed. &DFHW is the second of the variables CECI supplies, and it contains a command to write the contents of variable &DFHC (that is, your command) to the temporary storage queue named “Cltttt”, where “tttt” is the name of your terminal and two blanks precede the letters “CI”.
4. Execute this WRITEQ command (through the **command execution complete** screen). This stores your command on the queue.
5. If you want to save more than one command, repeat steps 2 to 4 for each.

When you want to execute the saved commands from the list, do the following:

1. Enter &DFHR on the command line and press ENTER. &DFHR is the last of the CECI-supplied variables, and it contains a command to read the queue that was written earlier. Execute this command; it brings the first (next) of the commands you saved into the variable &DFHC.
2. Then enter &DFHC on the command line and press ENTER. CECI replaces the command line with the value of &DFHC, which is your command. Press ENTER to execute your command.
3. Repeat these two steps, alternating &DFHR and &DFHC on the command line, until you have executed all of the commands you saved.

You can vary this procedure to suit your needs. For example, you can skip commands in the sequence by simply skipping step (2). You can change the options of the saved command before executing it in the same way as a command entered normally.

If you want to repeat execution of the saved sequence, you need to specify the option ITEM(1) on the first execution of the READQ command, in order to reposition your read to the beginning of the queue.

How CECI runs

There are several things you should know about how the interpreter works, in order to use it properly. These include:

- CECI sessions
- Abends
- Exception conditions
- Program control commands
- Terminal sharing
- Saving commands
- Shared storage

#

CECI sessions

The interpreter runs as a transaction, using programs supplied by CICS. It is conversational, which means that everything you do between the start of a session (entering CECI) and the end (PF3) is a single logical unit of work in a single task. This means that locks and enqueues produced by commands you execute remain for the duration of your session. If you read a record for update from a recoverable file, for example, that record is not available to any other task until you end CECI.

Abends

CECI executes all commands with the NOHANDLE option, so that execution errors do not ordinarily cause abends.

CECI also issues a HANDLE ABEND command at the beginning of the session, so that it does not lose control even if an abend occurs. Consequently, when you get one, CECI handles it and there is no resource backout. If you are doing a series of related updates to protected resources, you need to be sure that you do not do some of them and then find you cannot complete the others. If you find yourself in this situation, you can execute a SYNCPOINT ROLLBACK command or an ABEND command with the CANCEL option to remove the effects of your earlier commands on recoverable resources.

Exception conditions

For some commands, CECI may return exception conditions even when all specified options are correct. This occurs because, on some commands, CECI uses options that you do not specify explicitly. For example, the ASSIGN command always returns the exception condition INVREQ under CECI. Even though it may return the information you requested correctly, it will have attempted to get information from other options, some of which are invalid.

Program control commands

Because the interpreter is itself an application program, the interpretation of some program control commands may produce different results from an application program executing those commands. For example, ABEND command is intercepted, as noted above, unless you use the CANCEL option.

If you execute a LINK command, the target program executes, but in the environment of the interpreter, which may not be the one expected. In particular, if you modify a user display during a linked-to program, the interpreter will not be aware of the changes.

Similarly, if you interpret an XCTL command, CECI passes control to the named program and never gets control back, so that the CECI session is ended.

Terminal sharing

When the command being interpreted is one that uses the same screen as the interpreter, the command interpreter manages the sharing of the screen between the interpreter display and the user display.

The user display is restored:

- When the command being processed requires input from the operator
- When the command being processed is about to modify the user display
- When USER DISPLAY is requested

Thus, when a SEND command is followed by a RECEIVE command, the display sent by the SEND command appears twice, once when the SEND command is processed, and again when the RECEIVE command is processed. It is not necessary to respond to the SEND command, but, if a response is made, the interpreter stores it and redisplay it when the screen is restored for the RECEIVE command.

When the interpreter restores the user display, it does not sound the alarm or affect the keyboard in the same way as when a SEND command is processed. This is similar to EDF (see “Using EDF in single-screen mode” on page 620 for more information).

Shared storage: ENQ commands without LENGTH option

Normally, when you use the EXEC CICS ENQ command without the LENGTH
option, the effect is to specify as the resource a data area with a specific location
(address) in storage. Multiple tasks can enqueue on this resource and must refer to
the same location in storage. CECI is not able to emulate this behavior, because it
uses its own working storage, rather than shared storage.

If you execute an ENQ command in CECI without the LENGTH option, CICS
enqueues on an address within storage owned by the CECI task. Other tasks,

whether CECI or not, cannot enqueue on this same storage. CECI does not provide
support for using shared storage for its variables.

It is not possible to emulate the desired behavior by specifying the storage address
as the RESOURCE option, and adding the LENGTH option, when the ENQ
command is executed in a CECI task, then specifying the same storage address
without the LENGTH option in another CECI or non-CECI task. When the LENGTH
option is specified, CICS enqueues on the value of the resource rather than on its
location. CICS therefore regards the enqueues with and without the LENGTH option
as different enqueues, and the tasks are not serialized as intended.

When the LENGTH option is specified for the same ENQ command issued from
multiple tasks, the enqueue works as expected, because the location of the data
area (whether in storage owned by CECI or in other storage) does not matter when
the LENGTH option is specified.

Chapter 53. Using debuggers with CICS applications

CICS supports the use of workstation-based and host-based debuggers for isolating and fixing bugs, and for testing applications. This topic provides an overview of the high-level tasks you must perform before you can use a debugger with CICS applications, and tells you where to find more information:

Choose between a workstation-based and host-based debugger

When you debug an application program, you interact with the program through the debugging tools. For example, you may want to examine storage, set breakpoints, or step through your code. This interaction is a *debugging session*. In CICS, you can choose the environment in which you conduct your debugging session:

Workstation-based

A *debugger client* on the workstation provides a graphical user interface which you use to perform the debugging tasks. The debugger client communicates with a *debugger server* which runs on your CICS system, and interacts with the program that is being debugged.

For more information, see Chapter 54, “Debugging CICS applications from a workstation,” on page 663.

Host-based

A debugging tool running in your CICS system provides a terminal interface which you use to perform the debugging tasks. The debugging tool interacts directly with the application as it executes.

CICS supports Debug Tool for host-based debugging. For more information, see Chapter 55, “Using Debug Tool with CICS applications,” on page 665.

Different application programs may have different debugging requirements (for example, Java programs cannot be debugged in a host-based debugging session). CICS lets different users use workstation-based and host-based debugging concurrently in the same region.

Ensure that your application programs will be intercepted by the debugging tool, and that others will not

Even in a test or development system, most of your application programs will function correctly most of the time. And when you are debugging, you will probably want to focus on one application at a time. At the same time, your colleagues may want to debug different applications. So you will need a way to specify those programs in your CICS system that are to interact with your debugging session, and those that are to interact with other users' debugging sessions, while letting most programs in the system run normally.

Debugging profiles let you do all this — a debugging profile specifies a set of application programs which are to be debugged together. When you make a profile active, the programs it defines run under the control of the debugger, using a debugging session that you have specified. When you make the profile inactive, the programs run normally again, as do programs that are not referred to in debugging profiles. Debugging profiles also let you define the characteristics of the debugging session that you will use to debug a particular program.

For more information, see “Debugging profiles” on page 658.

Prepare your programs for interacting with a debugger

CICS supports application programs written in a variety of languages. The compiled language programs (COBOL, PL/I, C, C++, and Language Environment-enabled Assembler subroutines) run under the control of Language Environment; Java programs run in a Java virtual machine (JVM). Because there are, essentially, two different runtime environments for programs, there are two different ways to make your programs interact with the debugger.

- For compiled language programs, you must decide when you compile your programs that you want them to interact with the debugger, and specify the appropriate compiler options. See the compiler documentation for more information.
- For Java programs, you can decide at run time that you want them to interact with the debugger, and specify the appropriate JVM options. See the *CICS System Definition Guide* for more information.

Ensure that your CICS system is set up to support the debugging environment

When you have debugging profiles in your CICS system, there is an overhead in starting a program, even when all the profiles are inactive. This overhead, although small, is unlikely to be acceptable in a high-performance production system. In any case, you would not normally debug your applications in such a system. Therefore, the use of debugging profiles is optional, and if you want to use them, your system programmer will need to configure CICS accordingly.

For more information, see the *CICS System Definition Guide*.

Debugging profiles

A debugging profile specifies a set of one or more application programs which are to be debugged together. For example:

- All instances of program PYRL01 running in system CICS1
- All Java classes with names beginning “setBankAccount”
- All programs with names beginning “PYRL” run by user APPDEV02

CICS uses the following information in the debugging profile to decide if an instance of a program should run under the debugger's control. The parameters specify:

- The transaction under which the program is running
- The terminal associated with the transaction. You can specify the terminal identifier or the VTAM netname.
- The name of the program
- For COBOL programs, the name of the compile unit (the program or class name)
- For Java objects, and stateless CORBA objects, the class name
- For enterprise beans, the bean name
- For enterprise beans, and stateless CORBA objects, the method name
- The userid of the signed-on user
- The applid of the CICS region in which the transaction is running

Many of the parameters can be generic, allowing you to specify a set of values which begin with the same characters (for example, TRN0, TRN1, TRN2, TRNA, TRNB, ...)

Debugging profiles contain the following additional information:

Status

The status of the profile: *active* or *inactive*:

- When a profile is active, it is examined each time a program is started in a region for which debugging is required.

Note: If you change a profile while it is active, the changes take effect immediately: the next time a program is started, the changed parameters are used to decide if the program should run under the debugger's control.

- When a profile is inactive, it is ignored when a program is started.

Debugging display device settings

The debugging display device settings specify how you will interact with the debugger:

- For a Java program, you can use a debugging tool on a workstation
- For a compiled language program, you can use:
 - a 3270 terminal
 - a debugging tool on a workstation

The JVM profile name

For Java programs only, you can specify the JVM profile that will be used when a program is debugged

Debug Tool and Language Environment options

For compiled language programs only, you can specify options to be passed to Debug Tool and Language Environment when a program is debugged

You can create debugging profiles for the following sorts of program:

- Compiled language programs
- Java application programs
- Enterprise beans
- Stateless CORBA objects

The information stored in the profile is different for each type of program.

Profiles are stored in a CICS file which can be shared by more than one CICS region. A profile that is shared by several CICS regions is either active or inactive in all the regions: it cannot be active in some regions and inactive in others.

CICS provides a set of sample profiles which are optionally generated when your system is set up to use debugging profiles. You can use these profiles as a starting point for creating your own profiles.

Using debugging profiles to select programs for debugging

To select a program for debugging, you must create one or more debugging profiles. Each profile specifies a number of parameters that CICS uses to decide if an instance of a program should run under the debugger's control. Profiles can be active or inactive: if one of the active profiles matches the program instance, the program runs under the debugger's control. Inactive profiles are not examined when CICS starts a program. Profiles are inactive when they are created.

Table 46 on page 660 is an example which shows how parameters in the debugging profiles are used to select program instances for compiled language programs; Table 47 on page 660 shows how parameters in a debugging profile are used to select the program instance for a Java program.

Table 46. Examples of debugging profile parameters for compiled language programs

Debugging profile	Transaction	Terminal	Program	User	Applid
Profile 1	PRLA	T001	PYRL01	TESTER5	CICSTST2
Profile 2	PRLA	*	PYRL02	*	*
Profile 3	PRL*	*	*	*	CICSTST3

Table 47. Example of a debugging profile for a Java program

Debugging profile	Transaction	Bean	Method	User	Applid
Profile 4	PRLA	NewEmployee	setBasicSalary	TESTER5	CICSTST2

This is how each profile controls which programs run under the debugger's control:

- Profile 1** In this example, all the parameters in the table are specified explicitly: Program PYRL01 will run under the debugger's control only if all these conditions are satisfied:
- The transaction is PRLA
 - The transaction was started by terminal input from terminal T001
 - The transaction is being run by user TESTER5
 - The transaction is running in region CICSTST2
- Profile 2** In this example, some of the parameters in the table are *generic parameters*, specified as *; generic parameters of this type match all values. This profile specifies that every instance of program PYRL02 that runs under transaction PRLA will be under the debugger's control.
- Profile 3** This example contains another sort of generic parameter: PRL* matches all values that start with the characters 'PRL'. This profile specifies that every program that runs under a transaction whose ID starts with the characters 'PRL' in region CICSTST3 will be under the debugger's control.
- Profile 4** Method setBasicSalary will run under the debugger's control only if all these conditions are satisfied:
- The transaction is PRLA
 - The method is a method of bean NewEmployee
 - The transaction is being run by user TESTER5
 - The transaction is running in region CICSTST2

You should choose the parameters that you specify in your debugging profiles with care, to ensure that programs do not start under the debugger's control unexpectedly:

- If you can do so, specify values for all, or most, of the parameters, to restrict debugging to particular programs in particular circumstances. Use specific values rather than generic values where possible.
- Whenever possible, specify the userid and applid explicitly in each debugging profile.
- Although it is inadvisable to debug programs in a production region, there may be times when you need to do so. On these occasions, use a debugging profile in which all the parameters are specified explicitly.
- Activate debugging profiles only when you need to use them, and inactivate them immediately after use.

Using generic parameters in debugging profiles

You can supply generic values for many of the parameters in your debugging profiles. To specify generic parameters, use an asterisk (*) as a *wildcard character*. You can use the wildcard character on its own, or at the end of a parameter. Leaving a parameter blank is equivalent to specifying an asterisk. For example:

* matches all possible values

TR* matches TR, TRA, TRAA and TRAQ

TRA* matches TRA, TRAA and TRAQ, but not TR

When wildcards are used, a starting program may match more than one active profile. In this case, CICS selects the profile that is the best match, using the following principles:

- All parameters must match, either exactly, or when wildcards are considered.
- The best match is a profile that contains no wildcards.
- The next best matches are profiles that contain *. Within this grouping, the best matches are those that contain the smallest number of * characters, and the greatest number of explicitly specified characters.

For example, considering transaction TRAA:

- TRAA is the best possible match (all characters match)
- TRA* is a better match than TR*

It is advisable to avoid complex use of wildcards in your debugging profiles, as it is not always obvious which of many profiles will be the best match for a given program instance. However, should you need to do so, you can use the information in Figure 157 to work out exactly which of several profiles will be the best match.

For each field in turn:

1. Count the number of characters (excluding * but including trailing blanks) for each field (C)
2. Count the number of * characters (A)
3. Determine the length of the field (L)
4. Calculate M as $C - (L * A)$. Note that M may be negative.

For each profile in turn, sum the values of M for all the fields (R).

The profile with the greatest value of R is the best match. If two or more matching profiles have the same greatest value of R , CICS chooses one of them, basing its selection on the sequence in which the profiles were created.

Figure 157. The debugging profile matching algorithm

Chapter 54. Debugging CICS applications from a workstation

You can debug a CICS application using debugging tools that run on a workstation. There are two components to the debugging tools in this environment:

- The *debugger client*, which runs on the workstation. It is through the graphical user interface (GUI) provided by the debugger client that you interact with the application program. For example, you can use the debugger client to set breakpoints, to step through your program, and to examine the variables used by your program.
- The *debugger server*, which runs on the same system as the application program, and communicates with the debugger client.

You can debug the following sorts of CICS applications using a debugger client on a workstation:

- Applications written in a compiled language (COBOL, PL/I, C, C++)
- Language Environment-enabled Assembler subroutines
- Java applications running in a JVM
- Applications that use a combination of compiled language programs and Java programs

You cannot debug PLT programs using a debugger client on a workstation.

You can use the following as your debugger client:

WebSphere® Studio Enterprise Developer
WebSphere Studio Application Developer

For compiled languages and Language Environment-enabled Assembler subroutines, you can use the following products as your debugger server:

- Debug Tool

For Java programs, the debugger server is the Java Virtual Machine (JVM) executing in debug mode.

Preparing to debug applications from a workstation

Before you can debug CICS applications using a workstation, your system programmer must prepare your CICS region for debugging. See the *CICS System Definition Guide* for more information. You must then complete the following tasks:

1. Install a suitable debugger client in your workstation. You can use the following product as your debugger client:
WebSphere Studio Enterprise Developer
WebSphere Studio Application Developer

The documentation for these products contains the information you need to install and use them.

2. Create one or more *debugging profiles*. A debugging profile specifies which programs will run under the debugger's control.

Note: A debugging profile is *not* the same thing as a JVM profile. To debug a Java application, you need both profiles.

3. If you want to debug programs that are written in COBOL, PL/I, C or C++; or Language Environment-enabled Assembler subroutines, consider how you want to conduct your debug session, and compile your programs with the appropriate options. For more information, see *Debug Tool for z/OS User's Guide*.

4. If you want to debug a Java program, you must ensure that it runs in a Java virtual machine (JVM) that is enabled for debugging. To do this:
 - a. Create a JVM profile with parameters which enable the JVM for debugging. See *Java Applications in CICS* for more information.
 - b. Specify the JVM profile when you create a debugging profile for the Java program. If you do not specify the JVM profile, the JVM uses the profile specified in the PROGRAM definition.
5. Start the debugger client on your workstation.
6. If you are using WebSphere Studio as your debugger, set at least one breakpoint in your program.
7. Activate the debugging profiles that define the program instances that you want to debug. When you activate profiles for compiled language programs, you must define *debugging options* that specify the attributes of the debugging session that is started when the program runs.

When you have completed all these steps, the programs that you have selected at step 7 will run under the control of a debugger.

Chapter 55. Using Debug Tool with CICS applications

Debug Tool helps you test programs and examine, monitor, and control the execution of application programs. This chapter describes how you can use Debug Tool with CICS application programs. It contains the following topics:

- “About Debug Tool”
- “Preparing to debug applications with Debug Tool”

For detailed information about Debug Tool, see *Debug Tool for z/OS User's Guide*.

About Debug Tool

Debug Tool helps you test programs and examine, monitor, and control the execution of CICS application programs. You can debug the following sorts of CICS applications using Debug Tool:

- Applications written in a compiled language (COBOL, PL/I, C, C++)
- Language Environment-enabled Assembler subroutines
- Applications that use a combination of compiled language programs and Java programs. Debug Tool does not debug the Java portions of these applications

You cannot debug PLT programs using Debug Tool.

You can use Debug Tool in four ways:

Single terminal mode

Debug Tool displays its screens on the same terminal as the application

Dual terminal mode

Debug Tool displays its screens on a different terminal than the one used by the application

Batch mode

Debug Tool does not have a terminal, but uses a commands file for input and writes output to a log

Remote debug mode

Debug Tool works with a debugger client to display results on a workstation

For more information about Debug Tool, see *Debug Tool for z/OS User's Guide*.

Note: If you use Debug Tool in Single terminal mode, or Dual terminal mode, the terminal which is used by Debug Tool must be a local terminal in the region where the application is running: you cannot use a terminal on a terminal-owning region to interact with Debug Tool in an application-owning region.

Preparing to debug applications with Debug Tool

Before you can debug CICS applications using Debug Tool, your system programmer must prepare your CICS region for debugging. See the *CICS Application Programming Guide* for more information. You must then complete the following tasks:

1. Consider how you want to conduct your debug session, and compile your programs with the appropriate options. For more information, see *Debug Tool for z/OS User's Guide*.

2. Create one or more *debugging profiles*. A debugging profile specifies which programs will run under the debugger's control.
3. Activate the debugging profiles that define the program instances that you want to debug. When you activate the profiles, you must specify the display device with which you will interact with the debugger.

When you have completed all these steps, the programs that you have selected at step 3 will run under the control of Debug Tool.

Part 9. Appendixes

Appendix A. Using the phonetic conversion subroutine DFHPHN

You can use the DFHPHN macro to perform phonetic conversion of 16-character names (the DFHPHN macro is no longer supported). The result is a 4-byte phonetic equivalent. DFHPHN is supplied in CICSTS31.CICS.SDFHLOAD. For online code, DFHPHN takes the following parameters:

lang

A symbolic reference to a 1-byte argument indicating the programming language being used. This is X'F0' for COBOL, C/370, and assembler, and X'F1' for PL/I. If an error occurs during the processing of this request, then X'50' is returned in this argument; if no error occurs, X'00' is returned. This argument must be reset to indicate the programming language, before it can be reused.

name

a symbolic reference to the field that contains the 16-character name you want to convert.

phon

a symbolic reference to the field that contains the 4-byte phonetic equivalent of the name passed from the subroutine to the calling program.

For example;

Language	Command
COBOL C/370	<pre>CALL 'DFHPHN' USING lang name phon.</pre> <p>In the program prior to the main()[statement, code:</p> <pre>#pragma linkage(DFHPHN,0S) void DFHPHN();</pre> <p>In the program AFTER the main(){ statement, code:</p> <pre>DFHPHN(lang,name,phon);</pre>
PL/I Assembler	<pre>CALL DFHPHN (lang,name,phon); CALL DFHPHN, (lang,name,phon)</pre>

The phonetic code conversion subroutine (DFHPHN) also assists you to load and access data sets offline. The steps in creating such a data set would typically be:

1. Create the keys
 - a. Read a record from the source data set
 - b. Generate the “phon” using a call to the DFHPHN subroutine
 - c. Write the record to a temporary sequential data set
2. Sort the temporary data set on the 4-byte phonetic code
3. Load the key-sequenced VSAM data set
 - a. Read the sorted temporary data set
 - b. Write to the keyed data set

Appendix B. Migration for OS/VS COBOL programs

Runtime support for OS/VS COBOL programs is now withdrawn. These programs, which had runtime support in CICS Transaction Server for z/OS Version 2, cannot run under CICS Transaction Server for z/OS Version 3. This section is provided for migration purposes to help you upgrade OS/VS COBOL programs.

OS/VS COBOL programs must be upgraded to Language Environment conforming COBOL, and recompiled against a level of COBOL compiler supported by CICS. The *CICS Release Guide* lists the compilers supported by CICS. Enterprise COBOL for z/OS is the recommended compiler.

For information about writing CICS applications with Enterprise COBOL compilers, see Chapter 4, “Programming in COBOL,” on page 21.

Conversion to Enterprise COBOL

Many of the changes in the CICS-COBOL interface occur because Enterprise COBOL simplifies the procedures. This means that you do not need to use some CICS-specific OS/VS COBOL programming techniques. Of the changes described in this section, the only one that is mandatory is the replacement (removal) of all PROCEDURE DIVISION references to BLL cells.

Based addressing

Do not define and manipulate BLL cells. Review programs that use the CICS SET option and BLL cells, and make the following changes:

- Remove, from the linkage section, the entire structure defining BLL cells and the FILLER field. See Table 48 on page 672 for further information.
- Revise code that deals with chained storage areas to take advantage of the ADDRESS special register and POINTER variables.
- Change every SET(BLL cell) option in CICS commands to SET(ADDRESS OF A-DATA) or SET(A-POINTER) where A-DATA is a structure in the linkage section and A-POINTER is defined with the USAGE IS POINTER clause.
- Remove all SERVICE RELOAD statements.
- Remove all program statements needed in OS/VS COBOL to address structures in the linkage section longer than 4KB. A typical statement is:
ADD 4096, D-PTR1 GIVING D-PTR2.
- Remove artificial paragraph names where BLL cells are used to address chained storage areas.
- Review any program that uses BMS map data structures in its linkage section. The points to consider are:
 - In OS/VS COBOL programs, working storage is part of the compiled and saved program. Placing the maps in the linkage section reduces the size of the saved program, saving library space. In Enterprise COBOL, working storage is not part of the compiled program but is acquired dynamically.
 - If your map is in the linkage section, you can acquire and release the map storage dynamically with CICS GETMAIN and FREEMAIN commands. This helps you to optimize storage use, and can be useful in a long conversational transaction. This advantage of linkage section maps still applies in Enterprise COBOL.

- If your map is in the linkage section, you must issue a CICS GETMAIN command to acquire storage for the map. With OS/VS COBOL, you must determine the necessary amount of storage, which must be sufficient for the largest map in your map sets. This can be difficult to determine, and probably involves examining all the map assemblies. With Enterprise COBOL, use the LENGTH special register:

```
EXEC CICS GETMAIN
      SET(ADDRESS OF DATAREA)
      LENGTH(LENGTH OF DATAREA)
```

Table 48. Addressing CICS data areas in locate mode

OS/VS COBOL	Enterprise COBOL
WORKING-STORAGE SECTION. 77 LRECL-REC1 PIC S9(4) COMP. LINKAGE SECTION. 01 BLLCELLS. 02 FILLER PIC S9(8) COMP. 02 BLL-REC1A PIC S9(8) COMP. 02 BLL-REC1B PIC S9(8) COMP. 02 BLL-REC2 PIC S9(8) COMP. 01 REC-1. 02 FLAG1 PIC X. 02 MAIN-DATA PIC X(5000). 02 OPTL-DATA PIC X(1000). 01 REC-2. 02 PROCEDURE DIVISION. EXEC CICS READ UPDATE. . . SET(BLL-REC1A) LENGTH(LRECL-REC1) END-EXEC. ADD 4096 BLL-REC1A GIVING BLL-REC1B. SERVICE RELOAD REC-1. IF FLAG1 EQUAL X'Y' MOVE OPTL-DATA TO ... EXEC CICS REWRITE... FROM(REC-1) LENGTH(LRECL-REC1) END-EXEC.	WORKING-STORAGE SECTION. 77 LRECL-REC1 PIC S9(4) COMP. LINKAGE SECTION. 01 REC-1. 02 FLAG1 PIC X. 02 MAIN-DATA PIC X(5000). 02 OPTL-DATA PIC X(1000). 01 REC-2. 02 PROCEDURE DIVISION. EXEC CICS READ UPDATE . . SET(ADDRESS OF REC-1) LENGTH(LRECL-REC1) END-EXEC. IF FLAG1 EQUAL X'Y' MOVE OPTL-DATA TO ... EXEC CICS REWRITE . . FROM(REC-1) END-EXEC.

This table shows the replacement of BLL cells and SERVICE RELOAD in OS/VS COBOL by the use of ADDRESS special registers in Enterprise COBOL. If the records in the READ or REWRITE commands are fixed length, Enterprise COBOL does not require a LENGTH option. This example assumes variable-length records. After the read, you can get the length of the record from the field named in the LENGTH option (here, LRECL-REC1). In the REWRITE command, you must code a LENGTH option if you want to replace the updated record with a record of a different length.

Table 49 on page 673 shows the old and new methods of processing BMS maps in the linkage section. In this example, it is assumed that the OS/VS COBOL program has been compiled with the LONGLVL(1) option, and that the following map set has been installed:


```

MAPSET1 DFHMSD TYPE=DSECT,
          TERM=2780,LANG=COBOL,
          STORAGE=AUTO,
          MODE=IN

```

The new ADDRESS special register used in the example is described under “Using based addressing with COBOL” on page 27.

Table 49. Addressing BMS map sets in the linkage section

OS/VS COBOL	Language Environment conforming COBOL
WORKING-STORAGE SECTION. 77 FLD0 PIC X VALUE IS LOW-VALUE. LINKAGE SECTION. 01 BLLCELLS. 02 FILLER PIC S9(8) COMP. 02 BLL-DATAA PIC S9(8) COMP. 01 DATA1 COPY MAPSET1. PROCEDURE DIVISION. EXEC CICS GETMAIN LENGTH(1000) SET(BLL-DATAA) INITIMG(FLD0) END-EXEC.	WORKING-STORAGE SECTION. 77 FLD0 PIC X VALUE IS LOW-VALUE. LINKAGE SECTION. COPY MAPSET1. 01 MAP1 02 FILLER PIC X(12). 02 FILLER1L COMP PIC S9(4). . . 02 FIELD90 PIC X(20). PROCEDURE DIVISION EXEC CICS GETMAIN FLENGTH(LENGTH OF MAP1I) SET(ADDRESS OF MAP1I) INITIMG(FLD0) END-EXEC.

The highlighted material describes the contents of the MAP1I COBOL copybook.

Artificial assignments

Remove artificial assignments from an OCCURS DEPENDING ON object to itself. These are needed in OS/VS COBOL to ensure addressability.

Bibliography

The CICS Transaction Server for z/OS library

The published information for CICS Transaction Server for z/OS is delivered in the following forms:

The CICS Transaction Server for z/OS Information Center

The CICS Transaction Server for z/OS Information Center is the primary source of user information for CICS Transaction Server. The Information Center contains:

- Information for CICS Transaction Server in HTML format.
- Licensed and unlicensed CICS Transaction Server books provided as Adobe Portable Document Format (PDF) files. You can use these files to print hardcopy of the books. For more information, see “PDF-only books.”
- Information for related products in HTML format and PDF files.

One copy of the CICS Information Center, on a CD-ROM, is provided automatically with the product. Further copies can be ordered, at no additional charge, by specifying the Information Center feature number, 7014.

Licensed documentation is available only to licensees of the product. A version of the Information Center that contains only unlicensed information is available through the publications ordering system, order number SK3T-6945.

Entitlement hardcopy books

The following essential publications, in hardcopy form, are provided automatically with the product. For more information, see “The entitlement set.”

The entitlement set

The entitlement set comprises the following hardcopy books, which are provided automatically when you order CICS Transaction Server for z/OS, Version 3 Release 1:

Memo to Licensees, GI10-2559
CICS Transaction Server for z/OS Program Directory, GI10-2586
CICS Transaction Server for z/OS Release Guide, GC34-6421
CICS Transaction Server for z/OS Installation Guide, GC34-6426
CICS Transaction Server for z/OS Licensed Program Specification, GC34-6608

You can order further copies of the following books in the entitlement set, using the order number quoted above:

CICS Transaction Server for z/OS Release Guide
CICS Transaction Server for z/OS Installation Guide
CICS Transaction Server for z/OS Licensed Program Specification

PDF-only books

The following books are available in the CICS Information Center as Adobe Portable Document Format (PDF) files:

CICS books for CICS Transaction Server for z/OS

General

CICS Transaction Server for z/OS Program Directory, GI10-2586
CICS Transaction Server for z/OS Release Guide, GC34-6421
CICS Transaction Server for z/OS Migration from CICS TS Version 2.3, GC34-6425

CICS Transaction Server for z/OS Migration from CICS TS Version 1.3,
GC34-6423

CICS Transaction Server for z/OS Migration from CICS TS Version 2.2,
GC34-6424

CICS Transaction Server for z/OS Installation Guide, GC34-6426

Administration

CICS System Definition Guide, SC34-6428

CICS Customization Guide, SC34-6429

CICS Resource Definition Guide, SC34-6430

CICS Operations and Utilities Guide, SC34-6431

CICS Supplied Transactions, SC34-6432

Programming

CICS Application Programming Guide, SC34-6433

CICS Application Programming Reference, SC34-6434

CICS System Programming Reference, SC34-6435

CICS Front End Programming Interface User's Guide, SC34-6436

CICS C++ OO Class Libraries, SC34-6437

CICS Distributed Transaction Programming Guide, SC34-6438

CICS Business Transaction Services, SC34-6439

Java Applications in CICS, SC34-6440

JCICS Class Reference, SC34-6001

Diagnosis

CICS Problem Determination Guide, SC34-6441

CICS Messages and Codes, GC34-6442

CICS Diagnosis Reference, GC34-6899

CICS Data Areas, GC34-6902

CICS Trace Entries, SC34-6443

CICS Supplementary Data Areas, GC34-6905

Communication

CICS Intercommunication Guide, SC34-6448

CICS External Interfaces Guide, SC34-6449

CICS Internet Guide, SC34-6450

Special topics

CICS Recovery and Restart Guide, SC34-6451

CICS Performance Guide, SC34-6452

CICS IMS Database Control Guide, SC34-6453

CICS RACF Security Guide, SC34-6454

CICS Shared Data Tables Guide, SC34-6455

CICS DB2 Guide, SC34-6457

CICS Debugging Tools Interfaces Reference, GC34-6908

CICSplex SM books for CICS Transaction Server for z/OS

General

CICSplex SM Concepts and Planning, SC34-6459

CICSplex SM User Interface Guide, SC34-6460

CICSplex SM Web User Interface Guide, SC34-6461

Administration and Management

CICSplex SM Administration, SC34-6462

CICSplex SM Operations Views Reference, SC34-6463

CICSplex SM Monitor Views Reference, SC34-6464

CICSplex SM Managing Workloads, SC34-6465

CICSplex SM Managing Resource Usage, SC34-6466

CICSplex SM Managing Business Applications, SC34-6467

Programming

CICSplex SM Application Programming Guide, SC34-6468

CICSplex SM Application Programming Reference, SC34-6469

Diagnosis

CICSplex SM Resource Tables Reference, SC34-6470
CICSplex SM Messages and Codes, GC34-6471
CICSplex SM Problem Determination, GC34-6472

CICS family books

Communication

CICS Family: Interproduct Communication, SC34-6473
CICS Family: Communicating from CICS on System/390, SC34-6474

Licensed publications

The following licensed publications are not included in the unlicensed version of the Information Center:

CICS Diagnosis Reference, GC34-6899
CICS Data Areas, GC34-6902
CICS Supplementary Data Areas, GC34-6905
CICS Debugging Tools Interfaces Reference, GC34-6908

Other CICS books

The following publications contain further information about CICS, but are not provided as part of CICS Transaction Server for z/OS, Version 3 Release 1.

<i>Designing and Programming CICS Applications</i>	SR23-9692
<i>CICS Application Migration Aid Guide</i>	SC33-0768
<i>CICS Family: API Structure</i>	SC33-1007
<i>CICS Family: Client/Server Programming</i>	SC33-1435
<i>CICS Transaction Gateway for z/OS Administration</i>	SC34-5528
<i>CICS Family: General Information</i>	GC33-0155
<i>CICS 4.1 Sample Applications Guide</i>	SC33-1173
<i>CICS/ESA 3.3 XRF Guide</i>	SC33-0661

Books from related libraries

DL/I

If you use the CICS-DL/I interface, see the following manuals:

IMS Application Programming: Design Guide, SC27-1287
IMS: Application Programming: EXEC DLI Commands for CICS and IMS,
SC27-1288
IMS: Administration Guide: Database Manager, SC26-8725

DB2

For information about using DB2 SQL commands in a CICS application program, see the following manuals:

DB2 Universal Database for z/OS Administration Guide, SC26-9931
DB2 Universal Database for z/OS Application Programming and SQL Guide,
SC26-9933

Screen definition facility II (SDF II)

For information about Screen Definition Facility II, see the following manuals:

Screen Definition Facility II General Information, GH19-6114.
Screen Definition Facility II General Introduction Part 1, SH19-8128
Screen Definition Facility II General Introduction Part 2, SH19-8129

Screen Definition Facility II Primer for CICS/BMS Programs SH19-6118.
Screen Definition Facility II Preparing a Prototype, SH19-6458

Common programming interface

For information about the SAA interface, see the following manuals:

SAA CPI-C Reference, SC09-1308

Common Programming Interface Communications Reference, SC26-4399

SAA Common Programming Interface for Resource Recovery Reference,
SC31-6821

Common user access

For information about screens that conform to the CUA standard, see the following manuals:

SAA: Common User Access. Basic Interface Design Guide, SC26-4583

SAA: Common User Access. Advanced Interface Design Guide, SC26-4582

Programming languages

For information on programming in COBOL , see the following manuals:

Enterprise COBOL for z/OS: Customization Guide, GC27-1410

Enterprise COBOL for z/OS: Programming Guide, SC27-1412

Enterprise COBOL for z/OS: Compiler and Run-Time Migration Guide,
GC27-1409

For information on programming in C and C++, see the following manuals:

z/OS: C/C++ Compiler and Run-Time Migration Guide , GC09-4913

z/OS: C/C++ Programming Guide, SC09-4765

z/OS: C/C++ User's Guide, SC09-4767

z/OS: C/C++ Language Reference, SC09-4764

For information on programming in PL/I, see the following manuals:

Enterprise PL/I for z/OS: Programming Guide, SC27-1457

Enterprise PL/I for z/OS: Language Reference, SC27-1460

Enterprise PL/I for z/OS: Compiler and Runtime Migration Guide, GC27-1458

For information on programming in assembler language, see the following manuals:

Assembler H Version 2 Application Programming Guide, SC26-4036

Assembler H Version 2 Application Programming Language Reference,
GC26-4037

Teleprocessing Network Simulator (TPNS)

TPNS General Information, GH20-2487

TPNS Language Reference, SH20-2489

Language Environment:

z/OS: Language Environment Concepts Guide, SA22-7567

z/OS: Language Environment Run-Time Migration Guide, GA22-7565

z/OS: Language Environment Programming Guide, SA22-7561

z/OS: Language Environment Customization , SA22-7564

*z/OS: Language Environment Writing Interlanguage Communication
Applications*, SC28-1943

Miscellaneous books

2780 Data Transmission Terminal: Component Description, GA27-3005

8775 Display Terminal: Terminal User's Guide, GA33-3045

IBM InfoWindow 3471 and 3472 Introduction and Installation Planning Guide, GA18-2942

3270 Information Display System Data Stream Programmer's Reference, GA23-0059

3290 Information Display Panel Description and Reference, GA23-0021

8775 Display Terminal Component Description, GA33-3044

IBM CICS/ESA 3.3 3270 Data Stream Device Guide, SC33-0232

Determining if a publication is current

IBM regularly updates its publications with new and changed information. When first published, both hardcopy and BookManager® softcopy versions of a publication are usually in step. However, due to the time required to print and distribute hardcopy books, the BookManager version is more likely to have had last-minute changes made to it before publication.

Subsequent updates will probably be available in softcopy before they are available in hardcopy. This means that at any time from the availability of a release, softcopy versions should be regarded as the most up-to-date.

For CICS Transaction Server books, these softcopy updates appear regularly on the *Transaction Processing and Data Collection Kit* CD-ROM, SK2T-0730-xx. Each reissue of the collection kit is indicated by an updated order number suffix (the -xx part). For example, collection kit SK2T-0730-06 is more up-to-date than SK2T-0730-05. The collection kit is also clearly dated on the cover.

Updates to the softcopy are clearly marked by revision codes (usually a # character) to the left of the changes.

Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully.

You can perform most tasks required to set up, run, and maintain your CICS system in one of these ways:

- using a 3270 emulator logged on to CICS
- using a 3270 emulator logged on to TSO
- using a 3270 emulator as an MVS system console

IBM Personal Communications provides 3270 emulation with accessibility features for people with disabilities. You can use this product to provide the accessibility features you need in your CICS system.

Index

Numerics

- 10/63 magnetic slot reader 551
- 31-bit mode transaction 571
- 3262 printer 407
- 3270 bridge
 - ADS descriptor 467
- 3270 display 370
- 3270 family 387
 - attention keys 401
 - attributes, extended 394
 - base color 394
 - buffer 390
 - color, base 394
 - data stream 387
 - data stream orders 395
 - data stream, outbound 398
 - display characteristics 392
 - emulating 388
 - extended attributes 394
 - field attributes 392
 - field format, inbound 403
 - fields 392
 - inbound field format 403
 - input from 400
 - intensity 394
 - MDT 393
 - modified data tag 393
 - orders in data stream 395
 - outbound data stream 398
 - protection 393
 - reading from 403
 - screen fields 388
 - terminal, writing to 390
 - unformatted mode 404
 - write control character 391
 - writing to terminal 390
- 3270 Information Display System 127
- 3270 printer 408
 - options 409
- 3270 screen field 498
- 3289 printer 407
- 3290 display 541
 - character size 544
- 3601 logical unit 549
- 3770 batch data interchange logical unit 549
- 3770 batch logical unit 549
- 3790 batch data interchange logical unit 549

A

- abend 265
- ABEND command 274
- abend exit facility 273
- abend exit program 274
- abend exit routine 274
- abend user task, EDF 626
- abnormal termination recovery 273

- ACCEPT statement 24
- access to system information
 - EXEC interface block (EIB) 6
- ACCUM option 409, 483, 507
- ACK 369
- acknowledgment 369
- active partition 545
- ACTPARTN option 507, 545
- adding records 303
- ADDRESS command 5
- ADDRESS COMMAREA command 568
- addressing mode (AMODE)
 - options for CICS applications 100
- addressing of CICS areas 177
- ADS descriptor 467
- affinity 221
- AFTER option 558
- ALARM option 483
- ALLOCATE command 171
 - inhibit wait, NOSUSPEND option 171
- ALLOCERR condition 425
- alternate
 - index 289
 - key 289
- ALTPAGE value 513
- AMODE (addressing mode)
 - options for CICS applications 100
- APAK transaction 413
- APCG 166
- API
 - subset for DPL 443
- APOST option 75
- application program logical levels 33
- application programs
 - asynchronous processing 446
 - design 139
 - design for performance 165
 - distributed program link 435
 - distributed transaction processing 446
 - function shipping 434
 - installing 99
 - intercommunication considerations 433
 - logical levels 566
 - testing 604
 - transaction routing 434
 - using BMS map sets 105
 - writing 5
- area, dynamic storage 166
- argc 46
- argv 46
- ASIS option 499
- ASKTIME command 557
- assembler language
 - 31-bit mode 57
 - 64-bit registers 58
 - applications 168
 - CALL statement 60
 - programming techniques 57

- assembler language (*continued*)
 - restrictions 57
- Assembler language 89
 - DFHECALL macro 89
- assembly 67
- assembly, TYPE=DSECT 465
- ASSIGN command 5, 375, 413
 - DESTCOUNT option 534
 - MAPCOLUMN option 517
 - MAPHEIGHT option 517
 - MAPLINE option 517
 - MAPWIDTH option 517
 - MSR option 551
 - options 375
 - PAGENUM option 534
- asynchronous journal output 255
- asynchronous processing 433, 446
- AT option 558
- ATI 142, 367, 595
- ATNI 435
- attention field 553
- attention identifier 499
- ATTENTION key 369
- automatic task initiation 142, 367
- automatic transaction initiation 595
- AUTOPAGE option 510
- auxiliary storage
 - temporary data 598
- auxiliary temporary storage 158
- auxiliary trace 170
- AZI6 435

B

- backout of resources 257
- BAKR 57
- BASE option 478
- basic mapping support
 - assembling and link-editing physical map sets 128
 - assembling map 465
 - assembly, TYPE=DSECT 465
 - BMS support levels 453
 - complex fields 467
 - composite fields 468
 - copy facility 180
 - creating map 459
 - cursor position 488
 - cursor, finding the 500
 - data streams 176
 - data, moving to map 479
 - DFHASMVS procedure 128
 - DFHLNKVS procedure 128
 - DFHMAPS, procedure for installing maps 132
 - DFHMDF macro 459
 - DFHMDI macro 459, 461
 - DFHMSD macro 459, 462
 - DFHPSD, for defining partition sets 134
 - display, receiving data from 493
 - EOC condition 504
 - field, group 468
 - fields 456

- basic mapping support (*continued*)
 - fields, complex 467
 - fields, composite 468
 - fields, repeated 469
 - finding the cursor 500
 - full 454
 - group field 468
 - GRPNAME option 468
 - initializing output map 478
 - installing mapsets 126
 - installing partition sets 134
 - installing physical map sets 128
 - installing symbolic description map sets 129
 - invalid data 489
 - link-edit 465
 - macro 459
 - macros, rules for writing 463
 - map 455
 - map sets 466
 - map, assembling 465
 - map, creating 459
 - map, initializing output 478
 - map, moving data to 479
 - map, physical 465
 - map, symbolic 465
 - mapping input data 496
 - maps 176, 177, 180
 - maps, storage 477
 - MAPSET resource definition 465
 - MDT 498
 - message lengths, reducing 518
 - minimizing path length 517
 - minimum 453
 - modified data tag (MDT) 176
 - moving data to map 479
 - multimap screens 180
 - OCCURS option 469
 - output example 455
 - output map, initializing 478
 - page building operations 179
 - page routing operations 179
 - path length, minimizing 517
 - performance considerations 517
 - physical map 465
 - physical maps 126
 - preparing maps 125
 - PROGRAM resource definition 465
 - receiving data from display 493
 - reducing message lengths 518
 - repeated fields 469
 - rules for writing macros 463
 - screen copy 421
 - SEND MAP command 477
 - standard 453
 - storage for maps 477
 - support across platforms 454
 - symbolic description map sets for BMS 129
 - symbolic map 465
 - symbolic maps 126
 - terminals supported 454
 - TYPE=DSECT assembly 465

- basic mapping support (*continued*)
 - types of mapsets 126
 - upper case translation 499
 - using BMS map sets in application programs 105
 - using symbolic map sets in a program 130
- batch compilation 36
- batch data interchange 383
 - definite response 385
 - DEFRESP option 385
 - destination identification 385
 - ISSUE WAIT command 385
 - NOWAIT option 385
- BDAM 174
 - browsing operations 320
 - data sets 290, 319
 - exclusive control 319
 - updating operations 322
- BDI 407
- BGAM 366
- big COMMAREAs 189, 192, 209, 215
- blank fields 177
- block references 319
- block, Execution interface 45
- blocked-data set 291
- BMS 408
 - routing 417
- BMS commands 365
- BOTTOM command, CEBR transaction 637
- BRACKET option 380
- bracket protocol, LAST option 380
- bridge (3270)
 - ADS descriptor 467
- brightness 552
- browse operation
 - BDAM 320
- BROWSE TEMP STORAGE option, CEDF 626
- browsing 174
 - DELAY 174
 - records 298
 - SUSPEND 174
- BTS activities 207
- BUILDCHAIN 378

C

- C and C++ arguments 49
- C and C++ restrictions 45
- C language considerations
 - addressing EIB 50
 - data declarations 45
 - LENGTH option default 162
 - naming EIB fields 50
 - struct, symbolic description map set 126
- C++ considerations 51
- CALL statement
 - assembler language 60
- CANCEL command 557
- CARD option 385
- CBLCARD option 75
- CDUMP 46
- CEBR transaction 633

- CEBR transaction (*continued*)
 - body 635
 - BOTTOM command 637
 - browse transaction 633
 - CEBR initiation 633
 - COLUMN command 637
 - command area 635
 - displays 635
 - FIND command 637
 - GET command 637
 - header 635
 - initiation 633
 - LINE command 638
 - message line 635
 - PURGE command 638
 - PUT command 638
 - QUEUE command 638
 - security considerations 633
 - SYSID command 638
 - temporary storage browse 633
 - TERMINAL command 638
 - TOP command 639
 - transient data 639
- CECI transaction
 - about to execute command 644
 - ampersand (&) 650
 - body 646
 - command execution complete 645
 - command input 641
 - command input line 641
 - command line 641
 - command syntax check 643
 - ENQ commands 655
 - expanded area 650
 - information area 646
 - introduction 641
 - making changes 649
 - message line 646
 - PF key values area 646
 - program control 655
 - screen layout 641
 - shared storage 655
 - status area 642
 - terminal sharing 655
 - variables 650
- CECS transaction 648
- CEDF transaction 607, 608
 - abend user task 626
 - body 612
 - browse temporary storage 626
 - display register 627
 - displays 611
 - DPL 623
 - dual-screen mode 622
 - EDF transaction 608
 - functions 607
 - header 611
 - invoke CECI 627
 - invoking 608
 - modifying execution 624
 - non-terminal transactions 622

- CEDF transaction *(continued)*
 - options on function (PF) keys 626
 - overtyping displays 624
 - PF key 611
 - program labels 625
 - pseudoconversational programs 621
 - remote transactions 622
 - remote-linked programs 623
 - security 610
 - single-screen mode 620
- CESE transient data destination 115
- CESF, GOODNIGHT transaction 382
- chaining 370
- chaining of data 378
- CHANNEL
 - LINK command 565
 - option 187, 565, 569
 - RETURN command 565
 - XCTL command 565
- channels
 - as large COMMAREAs 189
 - basic examples 190
 - benefits of 215
 - compared to BTS activities 207
 - constructing 206
 - creating 195
 - current 196, 199
 - current, example, with LINK commands 196
 - current, example, with XCTL commands 198
 - data conversion 212
 - designing 205
 - discovering which containers a program's been passed 204
 - discovering which containers were returned from a link 204
 - dynamic and distributed routing 209, 219
 - LINK command 572
 - on LINK and XCTL commands 158
 - on RETURN commands 187
 - overview 189
 - read only containers 204
 - RETURN command 574
 - scope of 200, 202
 - typical scenarios
 - multiple interactive components 194
 - one channel—one program 192
 - one channel—several programs 193
 - several channels, one component 193
 - using from JCICS 209
- channels as large COMMAREAs 189, 192, 209, 215
- checkout, program 607
- CICS
 - testing environment 7
- CICS areas, addressing 177
- CICS dump utility program 278
- CICS option 75
- CICS printer 407
 - determining characteristics of 413
- CICS-key storage 579
- CICS-maintained table 291
- CICS-value data areas 162
- CICS DATAKEY option 156, 582
- CLASS option 427
- CLEAR
 - key 547
 - PARTITION AID value 547
 - PARTITION key 547
- CLEAR key 188
- client region 435
- CLOCK 46
- CMT 291
- CNOTCOMPL option 378
- COBOL
 - 31-bit addressing 25
 - addressing CICS data areas 27
 - blank lines 35
 - BMS data structures 671
 - calling subprograms 28
 - comma and semicolon delimiters 42
 - compiler options not used under CICS 24
 - compilers supported 21, 43, 53
 - DL/I CALL interface 26
 - elimination of SERVICE RELOAD statement 23
 - example of DFHNCTR call 353
 - global variables 42
 - LENGTH register 23
 - lower case characters 35
 - reference modification 41
 - REPLACE statement 35
 - reserved word table 23
 - restrictions 22, 167
 - RETURN CODE register 23
 - run unit 29, 33
 - sequence numbers 35
 - symbolic characters 42
 - WITH DEBUGGING MODE 28
- COBOL2 option 75
- COBOL3 option 75
- COLUMN
 - command, CEBR transaction 637
- COM assembler instruction 57
- comma and semicolon delimiters, COBOL 42
- command language translator 69, 73
 - APOST option 75
 - CBLCARD option 75
 - CICS option 75
 - COBOL2 option 75
 - COBOL3 option 75
 - CPSM option 76
 - DBCS option 76
 - DEBUG option 76
 - DLI option 76
 - EDF option 76
 - EPILOG option 76
 - EXCI option 76
 - FEPI option 77
 - FLAG option 77
 - GDS option 77
 - GRAPHIC option 77
 - LENASM option 78
 - LENGTH option 78
 - line numbers 71

command language translator *(continued)*

- LINECOUNT option 78
- LINKAGE option 78
- MARGINS option 78
- NATLANG option 79
- NOCBLCARD option 79
- NOCPSM option 79
- NODEBUG option 79
- NOEDF option 80
- NOEPILOG option 80
- NOFEPI option 80
- NOLENGTH option 80
- NOLINKAGE option 80
- NONUM option 80
- NOOPSEQUENCE option 80
- NOOPTIONS option 81
- NOPROLOG option 81
- NOSEQ option 81
- NOSEQUENCE option 81
- NOSOURCE option 71, 81
- NOSPIE option 81
- NOVBREF option 81
- NOXREF option 81
- NUM option 81
- OPMARGINS option 82
- OPSEQUENCE option 82
- options 73, 74
- OPTIONS option 82
- PROLOG option 82
- QUOTE option 82
- SEQ option 83
- SEQUENCE option 83
- SOURCE option 71, 83
- SP option 83
- SPACE option 83
- SPIE option 83
- SYSEIB option 83
- VBREF option 71, 84
- XOPTS keyword 73
- XREF option 84

command-level interpreter

- EIB 652
- ENTER key 646
- invoking 648
- messages display 652
- security considerations 641

command, SYNCPOINT 258

commands supported in C++ 51

COMMAREA 140, 156, 157, 166

- LINK command 565
- option 187, 565, 567, 569

COMMAREAs > 32K 189, 209, 215

common work area (CWA) 183

- protecting 184

compilation 67

compilers supported

- assembler 57
- C and C++ 43
- COBOL 21
- Language Environment 9
- PL/I 53

complex fields 467

components

- multiple, interactive 194
- one channel—several programs 193
- several channels, one component 193

composite fields 468

condition, exception 261

CONNECT PROCESS command 446

CONSISTENT option

- READ command 298
- READNEXT command 300
- READPREV command 300

CONSOLE option 385

constructing a channel 206

containers

- basic examples 190
- context, BTS or channel 208
- designing a channel 205
- discovering which a program's been passed 204
- discovering which were returned from a link 204
- overview 189
- read only 204
- using from JCICS 209

contention for resources 140

contention for terminal 367

context

- of containers, BTS or channel 208

control

- exclusive of BDAM 319
- of VSAM blocks 312

conversation partner 367

conversational programming 140, 181

CONVERSE command 181, 370, 381

copy facility

- BMS 180

COPY statements 86

copybook translation 86

counter name

- named counters 341

coupling facility data tables 292

coupling facility list structure

- current value 342

CPI Communications interface module, DFHCPLC 86

CPI Communications stub 447

CPI-C 433, 446

CPSM option 76

CQRY transaction 145

creating a channel 195

CSECT, adding to map assembly 132

CSNAP 46

CSPG transaction 417, 421, 509, 510

CSPP transaction 145

CTDLI 46

CTEST 46

CTLCHAR option 409, 410

CTRACE 46

current channel

- example, with LINK commands 196
- example, with XCTL commands 198
- overview 196, 199

CURSORM option 483, 488, 507

- cursor position 488
- cursor positioning, symbolic 488
- cursor-detectable field 552
- cursor, finding the 500
- CVDA 162, 281, 282
- CWA 183
- CWAKEY parameter 184

D

data

- chaining 378
- definition 167
- initialization 167
- passing to other program 567
- records 162
- storing within transaction 156
- data conversion 210
 - and channels 210
 - a SOAP example 214
 - why necessary 210
 - with channels 212
- data interchange block 71
- data sets 172
 - access from CICS application programs 296
 - batch data interchange 383
 - BDAM 290, 319
 - blocked 290
 - empty 289
 - sequential 175
 - user 160
 - VSAM 311
- data storing within transaction 156
- data streams
 - compressing 178
 - inbound 176
 - RA order 178
 - repeat-to-address orders (SBA) 178
 - SBA order 178
 - set buffer address order 178
- data tables
 - coupling facility 292
 - shared 291
- data, moving to map 479
- data, reading from a display 496
- DATA(24) 24
- DATA(31) 24
- database
 - DB2 323
 - DL/I 323
- DATAONLY option 177, 483, 485
- date field of EIB 6
- DB2 323
 - request processing 323
 - task related user exit 323
- DBCS option 76
- DCB interface of TCAM 383
- DDname list, in translator dynamic invocation 72
- DDS 473
- deadlock 169
 - prevention 312

- deadlocks 309
- DEBKEY option 321
- deblocking argument 319
- DEBREC option 320, 321
- DEBUG option 76
- debugging 607
- default
 - action for condition 261
- deferred journal output 256
- definite response protocol
 - terminal control 379
- DEFRESP option 182, 385
 - terminal control 379
- DELAY command 557, 558
- DELETE command 24
- DELETEDQ TD command 593
- DELETEDQ TS command 597
- deleting records 302
- DEQ command 561
- DEQUEUE command 417
- design considerations of applications
 - exclusive control of resources 169
- designator character 552
- designing a channel 205
- DESTCOUNT option 534
- DESTID option 385
- DESTIDLENG option 385
- destination identification 385
- detectable field 552
- device dependent support 473
- device-dependent maps 472
- DFH3QSS 7
- DFHAID 45
- DFHASMVS procedure 128, 130, 133
- DFHBMSCA 45, 481
- DFHBMSCA definitions 500
- DFHBMSUP 467
- DFHCOMMAREA 23, 567
- DFHCPLC 447
- DFHCPLC, CPI Communications interface module 86
- DFHCPLRR, SAA Resource Recovery interface
 - module 86
- DFHEAI, interface module for assembler 87
- DFHEAI0, interface module for assembler 87
- DFHEAP1\$, translator for assembler 72
- DFHECP1\$, translator for COBOL 72
- DFHEDF group 604
- DFHEDP1\$, translator for C 72
- DFHEIBLK 23
- DFHEIEND macro 81, 82
- DFHEIENT macro 62, 81, 82
- DFHEIRET macro 62, 76, 80
- DFHEISTG macro 81, 82
- DFHEITAL procedure 106, 109
- DFHEIVAR 23
- DFHELII 88, 89
- DFHELII, interface module for Language Environment
 - conforming compilers 87
- DFHEPP1\$, translator for PL/I 72
- DFHEXEC 89
- DFHEXTAL procedure 106

- DFHFCT macro 291
- DFHLNKVS procedure 128, 133
- DFHMAPS procedure 106, 131
- DFHMAPT
 - procedure for installing HTML templates 131
- DFHMDF macro 459
 - display characteristics 479
 - DSATTS option 479
 - MAPATTS option 479
- DFHMDI macro 459, 461
- DFHMIRS program 439
- DFHMSCAN utility program 100
- DFHMSD macro 459, 462
 - BASE option 478
 - STORAGE option 478
- DFHMSD, macro for assembling map sets 127
- DFHMSRCA 45, 551
- DFHNC001
 - default named counter pool 343
- DFHNCO macro
 - named counter options table 342
- DFHNCOPT
 - named counter options table 342
- DFHNCTR
 - example COBOL call with null pointers 353
- DFHPDI macro 543
- DFHPEP program 274
- DFHPSD macro 134, 543
- DFHRESP translator function 71, 262
- DFHURLDS 531
- DFHVALUE 71
- DFHYITDL procedure 106, 118
- DFHYITEL procedure 106, 118
- DFHYITPL procedure 106, 114
- DFHYITVL procedure 106, 111
- DFHYXTDL procedure 106, 118
- DFHYXTEL procedure 106, 118
- DFHYXTPL procedure 106, 114
- DFHYXTVL procedure 106, 111
- DFHZITCL procedure 106, 112
- DFHZITPL procedure 106, 115
- DIB 71
- direct terminal 535
- discovering which containers a program's been passed 204
- discovering which containers were returned from a link 204
- display
 - register, EDF 627
 - screens 188
- display characteristics 479
- DISPLAY statement 24
- display, reading from 496
- distributed application design 146
- distributed program link 565
 - client region 435
 - COMMAREA option 438
 - DPL API subset 443
 - exception conditions 444
 - independent syncpoints 440
 - options 436

- distributed program link *(continued)*
 - programming considerations 442
 - REMOTENAME option 438
 - REMOTESYSTEM option 439
 - server program 438
 - server region 435, 439
 - SYSID option 439
 - TRANSID option 439
- DL/I 25, 323
 - database operations 172
 - EXEC DLI interface 323
 - references 5
 - segment-search area (SSA) 172
 - syncpoints 258
- DLI 45, 77
- DLI option 76
- DLIUIB 45
- DOCTEMPLATE resource 326
- document
 - document template 326
 - TEMPLATE option 326
- document handler 325
- document template
 - CICS file 327
 - HFS file 327
 - in a CICS program 326, 327
 - in an exit program 328
 - in temporary storage 327
 - in transient data 327
- documents
 - creating 325
 - HTML 325
- DPL 258, 433, 435, 565, 623
- DSA 166
- DSATTS option 479
- DTP 433, 446
- DUMP TRANSACTION command 278
- DUPKEY condition 300
- dynamic
 - program 169
 - storage area 166
 - transaction backout program 274
 - transaction routing 571
- dynamic invocation of translator 72
- dynamic routing with channels 209, 219

E

- ECBLIST 562
- EDF 71, 76, 607
- EDF option 76
- EIB 45, 70, 261, 367
 - description 6
 - EIBCALEN field 568
 - EIBCOMPL field 371
 - EIBFN field 569
 - terminal control feedback 377
- empty data sets 289
- end-of-data indicator character 382
- ENDBR command 300
- ENQ command 171, 561

- ENQBUSY condition 171
- ENQUEUE command 417
- enqueueing
 - in a VSAM file 170
 - VSAM internal 170
- entry point, trace 277
- entry-sequenced data set (ESDS) 288
- EOC condition 378, 504
- EODI character 382
- EOS condition 378
- EPILOG option 76
- EQUAL option 297
- ERASE option 410, 483, 507
- ERASEAUP option 483, 507, 521
- ERDSA 103
- ESDS (entry-sequenced data set) 288
- events
 - monitoring point 277
- examples
 - channels, basic 190
 - CICS client program that constructs a channel 206
 - CICS server program that uses a channel 207
 - containers, basic 190
 - multiple interactive components 194
 - one channel—one program 192
 - one channel—several programs 193
 - several channels, one component 193
 - simple client program compared to a BTS activity 207
- exception condition
 - description 261
- exception conditions
 - HANDLE CONDITION command 266, 267
 - IGNORE CONDITION command 269
- exception trace entry point 277
- EXCI
 - CALL 447
 - communications 447
 - option 76
- EXCI - external call interface 433
- exclusive control of records
 - BDAM 319
 - VSAM 312
 - VSAM RLS 313
- exclusive resources 169
- EXEC DLI commands 46
- EXEC DLI interface 323
- EXEC interface block 70
- EXEC interface modules 86, 87, 120
- EXEC interface stubs 87
- EXEC SQL commands 323
- EXEC SQL interface 323
- EXECKEY 157, 184
- EXECKEY parameter 579
- execution diagnostic facility 71, 76, 607
- Execution interface block 45
- expiration time
 - specifying 558
- extended read-only DSA (ERDSA) 103
- external call interface (EXCI) 433
- External CICS interface (EXCI) 447

- extrapartition queues 594
- extrapartition transient data 160, 175
- EYUVALUE 71

F

- FEPI
 - references 5
- FEPI - Front End Programming Interface 433
- FEPI option 77
- FETCH 46
- field
 - blank 177
 - group 468
- fields
 - BMS 456
 - complex 467
 - composite 468
 - repeated 469
- file control
 - BDAM data sets 319
 - overview 287
 - VSAM data sets 311
- FIND command, CEBR transaction 637
- finding the cursor 500
- flag byte, route list 530
- FLAG option 77
- flip-flop mode 367
- floating maps 513
- FMH 380
 - inbound 380
 - option 380
 - outbound 380
- FMHPARM option 507
- FOR option 558
- formatted screen, reading from a 497
- FORMATTIME command 557
- FORMFEED option 411, 507
- FREE command 381
- FREELKB option 483
- FREEMAIN command 577
- FROM option 306, 483
- Front End Programming Interface (FEPI) 433
- FRSET option 483
- function (PF) keys, CEBR transaction 636
- function management header
 - description 380
- function shipping 434
- Function shipping 433
- function-ship 306
- functions, EDF 607

G

- GDDM 488
- GDS option 77
- generic delete 303
- generic key 297
- GENERIC option 297, 311
- GET command, CEBR transaction 637
- GETMAIN command 156

- GETMAIN command *(continued)*
 - CICSDATAKEY option 156, 582
 - INITIMG option 168, 577
 - NOSUSPEND option 577
 - SHARED option 156, 160, 577
 - TASKDATAKEY option 156
 - USERDATAKEY option 156, 582
- global user exits 581
- global variables, COBOL 42
- GOODNIGHT transaction, CESF 382
- GRAPHIC option 77
- group field 468
- GRPNAME option 468
- GTEQ option 297, 311

H

- half-duplex mode 367
- HANDLE ABEND command 262, 265, 273
- HANDLE ABEND LABEL with Assembler 57
- HANDLE AID command 499
- HANDLE CONDITION command 262, 265, 272
- HANDLE CONDITION ERROR command 269
- HOLD option 565
- HONEOM option 410
- horizontal tabs 412
- HTML templates
 - installing 131

I

- IBM Screen Definition Facility II (SDF II) 126
- ICTL (input format control) 57
- ICVR parameter 604
- identification
 - BDAM record 319
 - VSAM record 311
- IGNORE CONDITION command 262, 265, 269
- IGREQID condition 507
- IGYCCICS 23
- IMMEDIATE option 367, 380, 569
- IMS.RESLIB (IMS library) 123
- INBFMH condition 378
- inbound
 - data streams 176
- index, alternate 289
- indirect queues 595
- initializing output map 478
- INITIMG option 168, 577
- input
 - format control (ICTL) 57
- input data
 - chaining of 378
- input map, symbolic 495
- INPUTMSG option 565, 569, 571
- INQUIRE command 6
- INQUIRE TERMINAL command 377, 413
- INRTN option 546
- installing application programs
 - assembler-language 109
 - C 117

- installing application programs *(continued)*
 - COBOL 110
 - PL/I 114
 - using your own job stream 120
- installing assembler application programs
 - sample job stream for 109
- Installing HTML templates 131
- integrated CICS translator 67
- integrated translators 6, 67
- inter-transaction affinity
 - affinity life times 246
 - affinity transaction groups 247
 - caused by application generators 246
 - detecting 246
 - programming techniques 223
 - recommendations 224
 - relations and lifetimes 247
 - global relation 247
 - terminal relation 248
 - userid relation 250
 - safe programming techniques 224
 - the COMMAREA 225
 - the TCTUA 226
 - using BTS containers 229
 - using DEQ with ENQMODEL 228
 - using ENQ with ENQMODEL 228
 - suspect programming techniques
 - DELAY and CANCEL REQID commands 243
 - global user exits 222
 - INQUIRE and SET commands 222
 - POST command 244
 - RETRIEVE WAIT and START commands 240
 - START and CANCEL REQID commands 241
 - transient data 239
 - temporary storage data-sharing
 - temporary storage 236
 - unsafe programming techniques 229
 - the CWA 229
 - using DEQ 235
 - using ENQ 235
 - using LOAD PROGRAM HOLD 231
 - using shared storage 230
 - using task lifetime storage 232
 - using WAIT EVENT 234
- interactive debugging
 - CECI transaction 641
 - CECS transaction 648
 - CEDF transaction 607
- interactive problem control system 278
- intercommunication 433
- interface block, Execution 45
- interface modules
 - CPI Communications 86
 - EXEC 86
 - programs using EXEC CICS or EXEC DLI
 - commands 120
 - SAA Resource Recovery 86
 - using 108
- interface stubs, EXEC 87
- interleaving conversation with message routing 535
- interregion communication 447

- interrupting 369
- interval control 557
 - cancel interval control command 557
 - DELAY command 558
 - delay processing of a task 557
 - expiration time 558
 - POST command 558
 - specifying request identifier 559
 - START command 558
 - starting a task 557
- INTERVAL option 558
- INTO option 305
- intrapartition queues 593
- intrapartition transient data 160
- INVITE option 367
- invoking EDF 608
- INVPARTN condition 547
- INVPARTNSET condition 547
- INVREQ condition 507
- IPCS 278
- IRC 447
- iscics 7
- ISCINVREQ 435
- ISSUE ABORT command 384
 - CARD option 385
 - CONSOLE option 385
 - PRINT option 385
 - WPMEDIA1–4 option 385
- ISSUE ADD command 384
- ISSUE COPY command 369, 420
- ISSUE DISCONNECT command 369
- ISSUE END command 384
 - CARD option 385
 - CONSOLE option 385
 - PRINT option 385
 - WPMEDIA1–4 option 385
- ISSUE ERASE command 369, 384
- ISSUE NOTE command 384
- ISSUE PRINT command 420
- ISSUE QUERY command 384
- ISSUE RECEIVE command 384
- ISSUE REPLACE command 384
- ISSUE SEND command 384
 - CARD option 385
 - CONSOLE option 385
 - PRINT option 385
 - WPMEDIA1–4 option 385
- ISSUE WAIT command 384, 385
 - CARD option 385
 - CONSOLE option 385
 - PRINT option 385
 - WPMEDIA1–4 option 385

J

- JCICS
 - and channels 209
- JES 6, 423
- JES (job entry subsystem)
 - exits 427
 - input 427

- JES (job entry subsystem) *(continued)*
 - RESP and RESP2 options 425
 - retrieve data from JES spool 423, 424
 - spooler commands 425
- Job Entry Subsystem component of MVS 423
- journal
 - records 162, 255
- journal control
 - output synchronization 255
- journal identifier 257
- journal type identifier 257
- journaling 175, 255
- JOURNALNAME 257
- JOURNALNUM 257
- JTYPEID 257
- JUSFIRST option 522
- JUSLAST option 522
- JUSTIFY option 522

K

- key
 - alternate (secondary) 289
 - generic 297
 - hardware print 421
- key-sequenced data set (KSDS) 287
- KEYLENGTH option, remote data set 306
- keys
 - physical 319
- KSDS (key-sequenced data set) 287

L

- language considerations
 - assembler 57
- Language Environment 9
 - abend handling 12
 - under PL/I 55
- Assembler
 - conforming routines 58
 - non-conforming routines 60
- C and C++ 43
- callable services 10
- CEEBINT 19
- compilers supported 9
- condition handlers, user-written 12
- destinations, message and dump 11
- DLLs 15
- dump destination 11
- Dynamic Link Libraries 15
- HANDLE AID command 11
- HANDLE CONDITION command 11
- HLL user exit 19
- languages, mixing 13
- levels of support 9
- message destination 11
- mixing languages 13
- PL/I 54
- runtime options 16
 - and CICS LINK 18
 - determining which options were used 19

- Language Environment (*continued*)
 - runtime options (*continued*)
 - in child enclaves 18
 - RUWAPOL 11, 13
 - storage 11, 13
 - support, levels of 9
 - user-written condition handlers 12
 - VS COBOL II 26
- large COMMAREAs 189, 192, 209, 215
- LAST option 381, 484
 - bracket protocol 380
- LDC 549
- LDCMNEM option 550
- LDCNUM option 550
- LEASM option 78
- LENGERR condition 371
- LENGTH option 78, 162, 371
- LENGTH register, COBOL 23
- LENGTHLIST option
 - multiple dumps of task-related storage areas 279
- levels, application program logical 566
- library lookaside (LLA) 102
- light pen-detectable field 552
- LINE command
 - CEBR transaction 638
- line length on printing 410
- line traffic reduction 178
- LINECOUNT option 78
- LINK command 156, 165, 166, 566
 - CHANNEL option 565, 569
 - COMMAREA option 565, 567, 569
 - IMMEDIATE option 569
 - INPUTMSG option 565, 569
 - TRANSID option 569
- LINK commands, migrating to use channels 217
- link pack area (LPA) 102
- LINK PROGRAM 447
- link to program, expecting return 566
- link-edit 67, 71
- link-edit of map 465
- LINKAGE option 78
- LIST option 527
- LLA (library lookaside) 102
- LOAD command
 - HOLD option 565
- load libraries
 - support for secondary extents 108
- local copy key 420
- locale support 51
- locality of reference 167
- logging 175
- logical device component 549
- logical levels, application program 29, 33, 566
- logical messages, rules for 507
- logical record presentation 379
- logical unit of work (LUW)
 - database operations, control of PSB 172
 - description 141
 - recoverable resources 141
 - syncpoints used 257

- logical units (LUs)
 - facilities for 378
- lookaside transaction 543
- lower case characters 35
- LPA 102
- LU type 4
 - batch data interchange 384
 - device 368
 - logical record presentation 379
- LUs (logical units)
 - facilities for 378

M

- magnetic slot reader, 10/63 551
- main storage 166
 - temporary data 598
- main temporary storage 158
- map
 - BMS 455
 - creating 459
 - initializing output 478
 - link-edit 465
 - moving data to 479
 - sets 466
 - symbolic input 495
 - symbolic output 495
- map sets
 - Adding a CSECT 132
 - Using symbolic description map sets in a program 105
- MAPATTS option 479
- MAPCOLUMN option 517
- MAPFAIL condition 498, 504
- MAPHEIGHT option 517
- MAPLINE option 517
- MAPONLY option 177, 483, 484
- MAPPED option 525
- mapping input data 496
- maps
 - BMS 177, 180
 - device-dependent 472
 - floating 513
 - sets 168
- MAPSET option 483
- MAPSET resource definition 465
- mapsets
 - loading above the 16MB line 126
- MAPWIDTH option 517
- MARGINS option 78
- MASSINSERT option 174, 304
- MDT 176, 498
- MERGE statement 24
- message routing 527
- message title 533
- messages, undeliverable 532
- migrating COMMAREAs on RETURN commands 218
- migrating LINK commands that pass COMMAREAs 217
- migrating programs that use temporary storage to pass data 218

- migrating START data to use channels 218
- migrating XCTL commands that pass COMMAREAs 217
- migration 218
 - of LINK commands to use channels 217
 - of RETURN commands to use channels 218
 - of START data to use channels 218
 - of XCTL commands to use channels 217
 - to channels from COMMAREAs
 - exploiting the new function 216
 - without exploiting the new function 216
- mixed addressing mode transaction 571
- modernising COMMAREAs 189
- modified data tag 176, 498
- modifying execution, EDF 624
- modular program 168
- MONITOR command 277
- MONITOR POINT command 277
- monitoring application performance 277
- moving data to map 479
- MSGINTEG option 181
- MSR 551
- MSR option 507, 551
- multimap screens 180
- multipage outputs 179
- multithread testing 603
- multithreading 147
- MVS subspace 589
- MVS transaction 571
- MXT parameter 170

N

- named counters 341
 - CICS API 343
 - counter name 341
 - coupling facility list structure 342
 - current value 342
 - DFHNC001 343
 - DFHNCO macro 342
 - maximum value 342
 - minimum value 342
 - named counter fields 341
 - options table 342
 - overview 341
 - pools 342
- NATLANG option 79
- nested programs 38
- NLEOM option 408, 410, 507
- NOAUTOPAGE option 510
- NOCBLCARD option 79
- NOCPSM option 79
- NODE option 425
- NODEBUG option 79
- NOEDF option 80
- NOEDIT option 525
- NOEPILOG option 80
- NOFEPI option 80
- NOFLUSH option 514, 521
- NOHANDLE option 262, 268
- NOJBUFSP condition 171

- NOLENGTH option 80
- NOLINKAGE option 80
- non-CICS printer 407
- Non-CICS printer 417
- non-terminal transactions
 - EDF 622
- nonconversational programming 140
- NONUM option 80
- NOOPSEQUENCE option 80
- NOOPTIONS option 81
- NOPROLOG option 81
- NOQUEUE option 171
- NOSEQ option 81
- NOSEQUENCE option 74, 81
- NOSOURCE option 81
- NOSPACE condition 268
- NOSPIE option 81
- NOSUSPEND option 171
 - GETMAIN command 577
 - READ command 298
 - READNEXT command 300, 303
 - READPREV command 300, 303
 - WRITE command 304
- NOTRUNCATE option 371
- NOVBREF option 81
- NOWAIT option 385
- NOXREF option 81
- null parameters, example of DFHNCTR CALLs with 353
- null values, use of 179
- NUM option 81
- NUMREC option 303
- NUMSEGMENTS option
 - multiple dumps of task-related storage areas 279

O

- OCCURS option 469
- OPCLASS option 528
- OPEN command 24
- open transaction environment (OTE) 149
- operating system waits 171
- OPID option 528
- OPIDENT value 527
- OPMARGINS option 82
- OPSEQUENCE option 82
- OPSYN (equate operation code) 57
- option list, in translator dynamic invocation 72
- options
 - HANDLE CONDITION command 267
 - on function keys, EDF 626
- OPTIONS option 82
- OPTIONS(MAIN) specification 53
- OS/VS COBOL 671
- OTE, open transaction environment 149
- outboard controller 383
- outboard formatting 554
- output data, chaining of 378
- output map, initializing 478
- output map, symbolic 495
- OVERFLOW condition 514

- overlays 169
- overtyping EDF displays 624
- overview
 - dynamic routing with channels 209, 219

P

- PA key 413
- page break 514
- page building operations 179
- page fault 167
- page overflow 534
- page routing operations 179
- PAGENUM option 534
- PAGESIZE value 513
- paging
 - reducing effects 167
- PAGING option 409, 483, 489
- parameters
 - null 353
- partition sets
 - installing 133
 - loading above the 16MB line 126
- partition, active 545
- partitions 541
- partitions, defining 543
- PARTITIONSET option 544
- PARTN option 546
- partner, conversation 367
- partners, range of 446
- PARTNFAIL condition 547
- PARTNPAGE option 547
- passing control, anticipating return (LINK) 566
- passing data to other program 567
- pen-detectable field 552
- PERFORM command 6
- PERFORM DUMP command 278
- PF (program function) key 611, 635, 646
- physical keys 319
- physical map sets
 - installing 128
- PL/I language considerations
 - OPTIONS(MAIN) specification 53
 - restrictions 53
- PLT program 581
- plus 32K COMMAREAs 189, 209, 215
- POP HANDLE command 265, 274
- POST command 557, 558
- pre-translated code 119
- preprinted form 599
- presentation space 541
- PRGDLY option 532
- principal facility 142, 143
- print control bit 409
- print formatting 382
- print key 371, 420
- print key, Hardware 421
- PRINT option 385
- printed output, requests for 181
- printer
 - 3270 408

- printer (*continued*)
 - options 409
 - CICS 407
 - determining characteristics of 413
 - non-CICS 407
 - Non-CICS 417
 - SCS 410
- PRINTERCOMP option 412
- printing 407
 - CICS API considerations 418
 - line length on 410
 - START command 415
 - transient data 415
 - with BMS routing 417
- program
 - size 165
 - testing 607
- program control
 - linking to another program 566
 - passing data to another program 567
 - program logical levels 566
- program design
 - conversational 140, 181
 - nonconversational 140
 - pseudoconversational 140
- program labels in EDF 625
- PROGRAM option 273
- PROGRAM resource definition 465
- program storage 158
- programming models 139
- programming techniques
 - general 165
 - PL/I 53
- PROLOG option 82
- PROTECT option 181
- pseudoconversational programming 140
- PURGE command, CEBR transaction 638
- purge delay 532
- PURGE MESSAGE command 508, 527
- PUSH HANDLE command 265, 274
- PUT command, CEBR transaction 638

Q

- QBUSY condition 171
- QUERY SECURITY command 281
 - NOLOG option 282
 - RESCCLASS option 281
 - RESID option 281
 - RESTYPE option 281
- query transaction 145
- queue
 - temporary storage 597
- QUEUE command, CEBR transaction 638
- queues
 - extrapartition 594
 - intrapartition 593
 - transient data 593
- QUOTE option 82
- QZERO condition 595

R

- RACF 281
- range of partners 446
- RBA (relative byte address) 288, 311
- RDF 322
- READ command 24, 301
 - CONSISTENT option 298
 - NOSUSPEND option 298
 - REPEATABLE option 298
 - UNCOMMITTED option 298
- read only containers 204
- read-ahead queueing 368
- read-only DSA (RDSA) 103
- reading data from a display 496
- reading from a formatted screen 497
- reading records 296
- READNEXT command 298, 299
 - CONSISTENT option 300
 - NOSUSPEND option 300, 303
 - REPEATABLE option 300
 - UNCOMMITTED option 300
- READPREV command 299
 - CONSISTENT option 300
 - NOSUSPEND option 300, 303
 - REPEATABLE option 300
 - UNCOMMITTED option 300
- READQ TD command 171, 593
- READQ TS command 597
 - ITEM option 598
- RECEIVE command 181, 367, 369, 370, 381
 - MAPFAIL condition 504
- RECEIVE MAP command 496
 - ASIS option 499
- RECEIVE PARTN command 546
- record
 - identification 311, 319
 - locking 312
 - locking (RLS) 313
- record description field 322
- record locking 313
- record-level sharing (RLS)
 - accessing files in RLS mode 289
- records
 - adding 303
 - adding to BDAM data set 321
 - browsing 296
 - deleting 302
 - journal 255
 - length of 162
 - reading 296
 - updating 301
 - writing 301, 303
- recoverable resources 141
 - exclusive use 140
- recovery
 - of resources 169
 - problem avoidance 163
 - sequential terminal support 381
 - syncpoint 257
- reduction of line traffic 178
- reentrancy 147
- Reference modification 41
- reference set 168
- regression testing 603
- relative byte address (RBA) 288, 311
- relative record data set (RRDS) 288
- relative record number (RRN) 288, 311
- RELEASE 46
- RELEASE command
 - HOLD option 565
- RELEASE option 509
- RELTYPE keyword 291
- remote data set, KEYLENGTH option 306
- remote transactions, EDF 622
- remote-linked programs
 - DPL 623
 - EDF 623
- REMOTENAME option 438
- REMOTESYSTEM option 439
- REPEATABLE option
 - READ command 298
 - READNEXT command 300
 - READPREV command 300
- repeated fields 469
- REPLACE statement 35
 - translator action 35
- REQID option 300, 507, 532, 559
- request/response unit (RU) 378
- RESCLASS option 281
- RESETBR command 299
- RESID option 281
- residence mode (RMODE)
 - options for CICS applications 100
- resources
 - contention 140
 - control of 140
 - controlling sequence of access to 562
 - exclusive control of 169
 - exclusive use 140
 - recoverable 141, 169
- RESP and RESP2 options
 - for interface to JES 425
- RESP option 262, 268
 - deactivating NOHANDLE 267
- RESP value 261
- RESP2 option 262
- RESP2 value 261, 262
- restrictions
 - 31-bit mode 57
 - 64-bit registers 58
 - assembler language 57
 - PL/I 53
- RESTYPE option 281
- RETPAGE condition 491, 522
- RETRIEVE command 557, 562
- RETURN CODE register, COBOL 23
- RETURN command 158, 569
 - CHANNEL option 187, 565
 - COMMAREA option 187, 565
 - IMMEDIATE option 367, 380
 - INPUTMSG option 565, 569, 571
 - TRANSID option 142

- reverse interrupt 369
- REWRITE command 24, 301
- RIDFLD option 291, 298, 305
- RMODE (residence mode)
 - options for CICS applications 100
- ROUTE command 527
 - LIST option 527
 - page overflow 534
 - TITLE option 533
- route list 527
 - LIST option 530
 - segment chain entry format 530
 - standard entry format 530
- ROUTEDMSGS option 529
- routing terminals 535
- routing, Transaction 433
- RRDS (relative record data set) 288
- RRN (relative record number) 288, 311
- RTEFAIL condition 530
- RTESOME condition 530
- RU (request/response unit) 378
- rules for logical messages 507
- run unit in COBOL 29, 33
- runaway tasks 170
- RVI 369

S

- SAA Resource Recovery 258
- SAA Resource Recovery interface module, DFHCPLRR 86
- SAM 366
- scenarios
 - multiple interactive components 194
 - one channel—one program 192
 - one channel—several programs 193
 - several channels, one component 193
- scope of a channel
 - example, with LINK commands 200
 - example, with XCTL commands 202
 - overview 200
- screen copy, BMS 421
- screen field, 3270 498
- screen, reading from a formatted 497
- SCS
 - printer 410
- SCS input 413
- SDF II 459, 466
- SDF II (IBM Screen Definition Facility II) 126
- SEC system initialization option 282
- secondary extents, CICS load libraries 108
- security
 - CICS-defined resource identifiers 282
 - EDF 610
 - programming hints 282
 - record or field level 281
 - SEC system initialization option 282
 - SPCOMMAND resource type 282
- SEGMENTLIST option
 - multiple dumps of task-related storage areas 279
- selection field 553

- SEND command 181, 369, 370, 381
 - CNOTCOMPL option 378
 - CTLCHAR option 409
 - FMH option 380
 - INVITE option 367
 - LAST option 381
 - MSR option 551
- SEND CONTROL command 407, 485
- SEND MAP command 407, 477
 - ACCUM option 483, 507
 - ALARM option 483
 - CURSOR option 483, 488
 - DATAONLY option 483
 - ERASE option 483
 - ERASEAUP option 483, 521
 - FREEKB option 483
 - FROM option 483
 - LAST option 484
 - MAPONLY option 483
 - MAPSET option 483
 - NOFLUSH option 514, 521
 - PAGING option 483
 - SET option 483
 - TERMINAL option 483, 489
 - WAIT option 484
- SEND PAGE command 258, 508, 527
 - AUTOPAGE option 510
 - NOAUTOPAGE option 510
 - RELEASE option 509
- SEND PARTNSET command 545
- SEND TEXT command 407, 521
 - MAPPED option 525
 - NOEDIT option 525
- SEQ option 83
- sequence of access to resources, controlling 562
- SEQUENCE option 83
- sequential terminal support 381, 603
- server
 - program 438
 - region 435, 439
- SERVICE RELOAD
 - elimination, COBOL 23
- SESSBUSY condition 171
- SET
 - command 6, 489
 - option 305
- SET option 483, 489
- SETLOCALE 46
- shared control of records
 - VSAM RLS 313
- shared data tables 291
- SHARED option 156, 160
 - GETMAIN command 577
 - SHARED option 157
- shared storage 160
- sharing data across transactions 183
- SIGNAL condition 369
- simultaneous browse 300
- single-screen mode, EDF 620
- single-thread testing 603
- single-threading 147

- size, program 165
- SOURCE option 83
- SP option 83
- SPACE option 83
- space, presentation 541
- SPCOMMAND resource type 282
- SPIE option 83
- SPOLBUSY condition 427
- spool
 - commands 6
 - file 423
- SPOOLCLOSE command 423
- SPOOLOPEN
 - examples 430
- SPOOLOPEN command 418, 423
 - NODE option 425
 - TOKEN option 425
 - USERID option 425
- SPOOLREAD command 423
- SPOOLWRITE command 423
- SPURGE parameter 234
- SQL 5
- SQL interface, EXEC 323
- START command 24, 415, 557, 562
- START data, migrating to use channels 218
- STARTBR command 298
- static storage 168
- status flag byte, route list 530
- STOP statement 24
- storage
 - CICS-key 579
 - main 166
 - program 158
 - shareable 577
 - static 168
 - temporary 158
 - user 156
 - user-key 579
- storage area, dynamic 166
- storage control 577
- STORAGE option 23, 478
- storage protection 578
- struct, C/370 symbolic description map set 126
- stubs, EXEC interface 87
- subroutines 167
- subspace 589
- SUSPEND command 561
- suspend data set 598
- SVC99 46
- symbolic
 - input map 495
 - output map 495
- symbolic cursor positioning 488
- symbolic description map sets
 - using in a program 105
- synchronize action
 - journal output 255
- SYNCONRETURN option 440, 445
- SYNCPOINT command 258, 508
 - ROLLBACK option 274
- syncpointing 257, 259

- syncpointing, DPL 440
- SYSEIB option 83
- SYSID command, CEBR transaction 638
- SYSID option 439
- SYSIDERR 435
- SYSIN 70
- SYSARM, operand for assembling map sets 127
- SYSRINT 71
- SYPUNCH 70
- SYSTEM 46
- system information, access to 5
- system trace entry point 277

T

- tabs, horizontal 412
- tabs, vertical 412
- task control 561
 - sequence of access to resources 562
- task-related user exit 581
- TASKDATAKEY option 156, 581
- TASKDATALOC option 83, 156
- TCAM 383
 - DCB interface of 383
- TCTUA 186, 581
- TCTUAKEY 186, 581
- TCTUALOC 186
- techniques, programming 165
- template
 - CICS file 327
 - HFS file 327
 - in a CICS program 326, 327
 - in an exit program 328
 - in temporary storage 327
 - in transient data 327
- temporary data 597
- temporary storage
 - auxiliary 158, 598
 - browse transaction, CEBR 633
 - data 597
 - main 158, 598
 - queue 597
- temporary storage, used to pass data 218
- TERM option 473
- TERMIN value 528
- terminal
 - contention for 367
 - option 483
 - performance 176
 - sharing 655
 - support, sequential 381
 - wait 369
- TERMINAL
 - option 489
- TERMINAL command, CEBR transaction 638
- terminal control
 - bracket protocol, LAST option 380
 - break protocol 368
 - chaining of input data 378
 - chaining of output data 378
 - definite response 379

- terminal control (*continued*)
 - facilities for logical units 378
 - FMH, inbound 380
 - FMH, outbound 380
 - function management header (FMH) 380
 - interrupting 369
 - logical record presentation 379
 - map input data 496
 - print formatting 382
 - protocol, break 368
 - read-ahead queueing 368
 - table user area (TCTUA) 186
- Terminal control
 - commands 366
 - conversation partner 367
 - flip-flop mode 367
 - half-duplex mode 367
 - partner, conversation 367
- Terminal control commands 365
- TERMINAL option 489
- terminals, extended data stream 127
- TEST compiler option 24
- testing applications
 - multithread testing 603
 - preparing application table entries 604
 - preparing system table entries 604
 - preparing the system 604
 - regression testing 603
 - sequential terminal support 603
 - single-thread testing 603
 - using sequential devices 381, 603
- threadsafe programs 149
 - effect of
 - ADDRESS CWA 152
 - EXTRACT EXIT 152
 - GETMAIN SHARED 152
- time field of EIB 6
- TIOATDL value 525
- TITLE option 533
- title, message 533
- TOKEN option 306, 425
- TOP command, CEBR transaction 639
- trace
 - description 276
 - trace entry point 277
- TRANISO 587
- transaction
 - affinity 434, 557, 562, 565, 577, 586
 - deadlock 307
 - routing 433, 434
 - routing, dynamic 571
- transaction affinity
 - inter-transaction affinity 222
 - transaction-system affinity 222
- transaction identifier
 - CEBR 633
 - CECI 648
 - CEDF transaction 607
- transaction isolation 578
- transaction work area 156
- transaction-system affinity 222

- transactions
 - conversational 140
 - nonconversational 140
 - pseudoconversational 140
- TRANSID option 439, 569
- transient data 415
 - extrapartition 160, 175
 - intrapartition 160
 - queue 160, 239
- transient data control
 - automatic transaction initiation (ATI) 595
 - queues 593, 594
- translation 6, 67
- translator
 - dynamic invocation of 72
 - integrated with Language Environment-conforming compilers 67
- translator data sets 69, 73
- trigger field 551
- TRUNC option 24
- TS queue 236
- TST TYPE=SHARED 237
- TWA 156
- TWASIZE option 156
- TYPE=DSECT assembly 465

U

- UMT 292
- UNCOMMITTED option
 - READ command 298
 - READNEXT command 300
 - READPREV command 300
- undeliverable messages 532
- unit of compilation, COBOL
 - start of, nested programs 38
 - submitting to translator 39
 - translator options for 36
- unit of work 258
- UNLOCK command 304
- UNTIL option 558
- UOW 258
- update operation, BDAM 322
- UPDATE option 301
- updating records 301
- upgrade set 289
- upper case translation in BMS 499
- user
 - data sets 160
 - storage 156
 - trace entry point 277
- user-key storage 579
- user-maintained table 292
- user-replaceable module 581
- USERDATAKEY option 156, 582
- USERID option 425, 427
- using channels from JCICS 209

V

- validity of reference 167
- variables, CECI/CECS 650
- VBREF option 84
- vertical tabs 412
- viewport 541
- virtual lookaside facility (VLF) 102
- virtual storage 166
- virtual storage environment 165
- VLF (virtual lookaside facility) 102
- VOLUME option 385
- VOLUMELENG option 385
- VSAM
 - data sets 173, 311
 - enqueues 170
 - MASSINSERT option 174
 - processor overhead 173
- VTAM 366, 367

W

- WAIT EVENT command 234, 557
- WAIT EXTERNAL command 235, 561
- WAIT JOURNALNUM command
 - synchronize with journal output 255
- WAIT option 256, 369, 484
- WAIT TERMINAL command 369
- wait, terminal 369
- WAITCICS command 235, 561
- waits, operating system 171
- WITH DEBUGGING MODE 28
- working set 167
- working storage 23, 44, 57
- WPMEDIA1–4 option 385
- WRITE command 24, 303
 - NOSUSPEND option 304
- WRITE JOURNALNAME command 171, 255
- WRITE JOURNALNUM command 171, 255
 - create a journal record 255
- WRITEQ TD command 593
- WRITEQ TS command 597
- writing records 301, 303
- WRKAREA parameter 183

X

- XCTL command 156, 158, 165, 166
 - CHANNEL option 565, 569
 - COMMAREA option 565, 567
 - INPUTMSG option 565, 569
- XCTL commands, migrating to use channels 217
- XOPTS keyword 73
- XPCREQ global user exit 435, 439
- XREF option 84
- XTC OUT exit 178
- XTSREQ, global user exit 237

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

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

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply in the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM United Kingdom Laboratories, MP151, Hursley Park, Winchester, Hampshire, England, SO21 2JN. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. A current list of IBM trademarks is available on the Web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other product and service names might be trademarks of IBM or other companies.

Sending your comments to IBM

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book.

Please limit your comments to the information in this book and the way in which the information is presented.

To ask questions, make comments about the functions of IBM products or systems, or to request additional publications, contact your IBM representative or 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.

You can send your comments to IBM in any of the following ways:

- By mail, to this address:

IBM United Kingdom Limited
User Technologies Department (MP095)
Hursley Park
Winchester
Hampshire
SO21 2JN
United Kingdom

- By fax:
 - From outside the U.K., after your international access code use 44–1962–816151
 - From within the U.K., use 01962–816151
- Electronically, use the appropriate network ID:
 - IBMLink: HURSLEY(IDRCF)
 - Internet: idrcf@hursley.ibm.com

Whichever you use, ensure that you include:

- The publication title and order number
- The topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.



Product Number: 5655-M15

SC34-6433-06



Spine information:



CICS TS for z/OS

CICS Application Programming Guide

Version 3
Release 1