CICS Transaction Server for z/OS
Version 5 Release 1

Application Programming Guide

SC34-2844-02

CICS Transaction Server for z/OS
Version 5 Release 1

Application Programming Guide

SC34-2844-02

Note
Before using this information and the product it supports, read the information in "Notices" on page 785.

This edition applies to Version 5 Release 1 of CICS Transaction Server for z/OS (product number 5655-Y04) and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright IBM Corporation 1989, 2013.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

What this manual is about xi
Who should read this manual xi
What you need to know to understand this manual xi
How to use this manual xi
Location of topics in the information center . . . xi
Notes on terminology xii
What is not covered in this manual xii

Changes in CICS Transaction Server for z/OS, Version 5 Release 1 xiii

Part 1. Writing CICS applications . . 1

Chapter 1. Writing CICS applications . . 3
What is a CICS application? 3
CICS programs, transactions and tasks. 3
CICS programming 4
CICS programming commands 5
EXEC interface block (EIB). 5
Translation 6
Testing for CICS 6
CICS programming roadmap 7
Identifying Programming Interfaces. 7

Part 2. Programming languages and Language Environment 9

Chapter 2. Programming languages and Language Environment. 11
Language Environment callable services 12
Language Environment abend and condition handling 13
Language Environment storage 14
Mixing languages in Language Environment . . . 15
Dynamic Link Libraries (DLLs) 17
Defining runtime options for Language Environment 17
CEEBXITA and CEECSTX user exits 20
CICSVAR: CICS environment variable 20
CEEBINT exit for Language Environment 22

Chapter 3. Developing COBOL applications 23
COBOL programming restrictions and requirements 24
Language Environment CBLPSHPOP option . . 26
Using the DL/I CALL interface. 27
VS COBOL II programs 28
Using based addressing with COBOL. 29
Calling subprograms from COBOL programs . . . 30
Flow of control between programs and subprograms 31
Rules for calling subprograms 33
COBOL2 and COBOL3 translator options 36
CICS translator actions for COBOL programs . . . 37
Batch compilation for COBOL programs. 39
Nested COBOL programs. 41

Chapter 4. Programming in C and C++ 45
C and C++ programming restrictions and requirements 46
Passing arguments in C and C++ 50
Accessing the EIB from C and C++ 51
Locale support for C and C++ 52
XPLink and C and C++ programming 52
XPLink uses X8 and X9 mode TCBs 53
Passing control between XPLink and non-XPLink objects 53
Global user exits and XPLink 54

Chapter 5. Developing PL/I applications 55
PL/I programming restrictions and requirements. . 55
Language Environment coding requirements for PL/I applications 57
Fetched PL/I routines 58

Chapter 6. Developing Assembler language applications 61
Assembler language programming restrictions and requirements 61
Coding the EXEC CICS assembler interface. . . . 65
Language Environment coding requirements for assembler language applications 69
Calling assembler language programs 71
Extensions to dynamic storage 73
Developing AMODE(64) assembler language programs 75

Part 3. Translating, compiling, installing and testing application programs 77

Chapter 7. Translation and compilation 79
The integrated CICS translator 79
Using the integrated CICS translator 80
Specifying CICS translator options. 80
The translation process 81
The CICS-supplied translators 84
Dynamic invocation of the separate translator . . 84
Using a CICS translator 94
Defining translator options 95
Translator options table 96
Using COPY statements 98
The CICS-supplied interface modules. 98
Using the EXEC interface modules for AMODE(24) and AMODE(31) applications 99

© Copyright IBM Corp. 1989, 2013

Example assembler language program with LEASM 102
Using the EXEC interface modules for AMODE(64) applications 104

Chapter 8. Deploying applications to a platform. 105
Installing an application in a platform 105
Getting started with cloud enablement 107

Chapter 9. Installing application programs 117
Program installation steps 118
Using dynamic program LIBRARY resources . . . 118
Examples of using dynamic LIBRARY resources 119
Defining MVS residence and addressing modes 126
Establishing the addressing mode of a program 126
CICS address space considerations 127
Making programs permanently resident . . . 128
Running applications in the link pack area . . . 128
Running application programs in the read-only DSAs 129
Assembler 129
C and C/++. 131
COBOL 131
PL/I 132
Using BMS map sets in application programs . . 132
Using the CICS-supplied procedures to install application programs. 133
Installing programs in load library secondary extents 134
Including the CICS-supplied interface modules . . 135
Translating, assembling, and link-editing assembler language application programs 135
Installing COBOL application programs 137
Sample JCL to install COBOL application programs. 138
Installing PL/I application programs 141
Installing C application programs 141
Sample JCL to install C application programs 143
Using your own job streams 145

Chapter 10. Installing map sets and partition sets 149
Installing map sets 150
Types of map sets 150
Installing physical map sets 152
Installing symbolic description map sets . . . 154
Installing physical and symbolic description maps together 155
Installing partition sets 158
Defining programs, map sets, and partition sets to CICS 159

Chapter 11. Testing applications . . . 161
Preparing the application for testing. 162
Preparing the system for testing 162

Chapter 12. Execution diagnostic facility (EDF). 165
Restrictions when using EDF 166
What does EDF display? 168
The header 169
The body. 169
Testing programs using EDF 175
Interrupting program execution 175
Using EDF in single-screen mode. 176
Using EDF in dual-screen mode 178
EDF and remote transactions 178
EDF and non-terminal transactions 179
EDF and DTP programs. 179
Stopping EDF 180
Using EDF to change information 180
EDF responses 182
Using EDF menu functions. 182

Chapter 13. Temporary storage browse (CEBR) 191
Using the CEBR transaction 191
What does the CEBR transaction display? 192
Using the CEBR function keys. 193
Using the CEBR commands 194
Using the CEBR transaction with transient data 197

Chapter 14. Command-level interpreter (CECI). 199
What does CECI display? 199
The command line 199
The status line 200
The screen body 203
The message line 204
CECI options on function keys 204
Using CECI 205
Defining variables. 206
Saving commands. 207
How CECI runs 208
Shared storage: ENQ commands without LENGTH option. 209

Chapter 15. Preparing to use debuggers with CICS applications . . 211
Debugging profiles 212
Using debugging profiles to select programs for debugging 213
Using generic parameters in debugging profiles 215

Chapter 16. Debugging CICS applications from a workstation . . . 217
Preparing to debug applications from a workstation 217

Chapter 17. Using Debug Tool with CICS applications 219
About Debug Tool. 219
Preparing to debug applications with Debug Tool 219

Part 4. CICS application programming techniques 221

Chapter 18. Application design 223

Pseudoconversational and conversational design 223
Terminal interruptibility 225
How tasks are started 225
Which transaction? 226
Separating business and presentation logic . . . 229
Multithreading: Reentrant, quasi-reentrant, and threadsafe programs 230
Quasi-reentrant application programs 231
Threadsafe programs 232
Making applications threadsafe 235
CONCURRENCY(REQUIRED) programs . . . 238
OPENAPI programs 239
Using the FORCEQR system initialization parameter 242
Non-reentrant programs 242
Storing data in a transaction 243
Transaction work area (TWA) 243
User storage 244
COMMAREA in LINK and XCTL commands 244
Channels in LINK and XCTL commands . . . 245
Program storage 245
Temporary storage queues 245
Intrapartition transient data 248
GETMAIN SHARED command 248
Your own data sets 249
Lengths of areas passed to CICS commands . . . 249
Minimizing errors 250
Protecting CICS from application errors . . . 250
Testing applications 251
Non-terminal transaction security 251
Design for performance 252
Program size 253
Virtual storage 254
Exclusive control of resources 257
Operational control 258
Operating system waits 259
The NOSUSPEND option 259
Efficient sequential data set access 260
Efficient logging 261
Sharing data across transactions 261
Using the common work area (CWA) 261
Using the TCTTE user area (TCTUA) 264
Using the COMMAREA in RETURN commands 265
Using a channel on RETURN commands . . . 266
Using the display screen to share data 266
Transferring data between programs using channels 267
Channels: quick start 267
Using channels: some typical scenarios 270
Creating a channel 273
The current channel 274
The scope of a channel 279
Discovering which containers were passed to a program 282
Discovering which containers were returned from a link 283
CICS read-only containers 283
Designing a channel: Best practices 284
Constructing and using a channel: an example 285
Channels and BTS activities 286
Using channels from JCICS 288
Dynamic routing with channels 288
Data conversion 289
Benefits of channels 295
Migrating from COMMAREAs to channels . . 296
Program control 300
Program linking 301
Passing data to other programs 302
Using mixed addressing modes 306
Using LINK to pass data 307
Using RETURN to pass data 309
Affinity 312
Types of affinity 313
Programming techniques and affinity 314
Programming techniques that avoid affinity . . 316
Programming techniques that create affinities 320
Programming techniques that might create affinities 327
Detecting inter-transaction affinities 337
Duration and scope of inter-transaction affinities 338
Recovery design 345
Journaling 345
Syncpointing 348
Dealing with exception conditions 349
Default CICS exception handling 350
Handling exception conditions by inline code 351
Modifying default CICS exception handling . . 354
Using the HANDLE CONDITION command 355
Using the HANDLE CONDITION ERROR command 357
Using the IGNORE CONDITION command . . 358
Using the HANDLE ABEND command . . . 359
Using PUSH HANDLE and POP HANDLE commands 360
Abnormal termination recovery 361
Creating a program-level abend program or routine 363
Retrying operations 364
Trace 365
Monitoring application performance 366
Dump 367
The **QUERY SECURITY** command 368
Using the QUERY SECURITY command . . . 368
CICS intercommunication 369
Transaction routing 370
Function shipping 371
Distributed program link (DPL) 371
Asynchronous processing 383
Distributed transaction processing (DTP) . . . 383
Common Programming Interface Communications (CPI Communications) . . . 384
External CICS interface (EXCI) 384

Part 5. Data mappings 387

Chapter 19. Mapping and transforming application data and XML 389
The CICS XML assistant 389
DFHLS2SC: high-level language to XML schema conversion 390
DFHSC2LS: XML schema to high-level language conversion 396
Mapping levels for the CICS assistants 404
High-level language and XML schema mapping 408
Variable arrays of elements 435
Support for XML attributes 439
Support for <xsd:any> and xsd:anyType . . . 442
Support for <xsd:choice> 444
Support for <xsd:sequence> 446
Support for substitution groups 446
Support for abstract elements and abstract data types 448
How to handle variably repeating content in COBOL 450
Support for variable-length values and white space 453
Generating mappings from language structures . . 455
Generating mappings from an XML schema . . . 458
Transforming application data to XML 460
Transforming XML to application data 461
Querying XML from an application 462
Handling XML by data type 463
Handling <xsd:any> data types 464
Validating XML transformations 465

Part 6. Business services and bundles 467

Chapter 20. Creating business services from CICS applications . . . 469
Service Component Architecture (SCA) 469
SCA composites and wiring 470
Best practices for creating and deploying composites 472
Creating a channel-based service 473
Creating an XML-based service 475
CICS processing of services. 476
Troubleshooting SCA problems 478

Part 7. File control 479

Chapter 21. Understanding file control 481
VSAM data sets: KSDS, ESDS, RRDS 481
Accessing files in RLS mode 483
Identifying VSAM records 484
Key 484
Relative record number (RRN), relative byte address (RBA) and extended relative byte address (XRBA) 485
Upgrading to extended addressing for ESDS . . . 486
Locking of VSAM records in recoverable files . . 488
RLS Record level locking 489
Exclusive locks and shared locks 489
Lock duration 490
Active and retained states for locks 491
BDAM data sets 492
Identifying BDAM records 493
CICS shared data tables 495
Coupling facility data tables 495
Techniques for sharing data 497
Transaction deadlocks 500
VSAM-detected deadlocks (RLS only) 501
Rules for avoiding deadlocks 502
File control operations 502
Using CICS commands to read records 502
Using CICS commands to update records . . . 510
Using CICS commands to delete records . . . 512
Updating and deleting records in a browse (VSAM RLS only) 513
Using CICS commands to add records 514
Efficient data set operations 517

Part 8. Terminal control 519

Chapter 22. Terminal access method support 521

Chapter 23. Terminal control commands 523
Send/receive mode 523
Contention for the terminal. 524
RETURN IMMEDIATE 524
Speaking out of turn 524
Interrupting 525
Terminal waits 525

Chapter 24. Using data transmission commands 527
What you get on a RECEIVE 527

Chapter 25. Device control commands 529

Chapter 26. Terminal device support 531

Chapter 27. Finding out about your terminal 535
EIB feedback on terminal control operations . . . 536

Chapter 28. Using SNA 539
Chaining input data 539
Chaining output data. 539
Handling logical records. 540
Response protocol 540
Using function management headers 541
Inbound FMH 541
Outbound FMH 541
Preventing interruptions with bracket protocol . . 541

Chapter 29. Using sequential terminal support 543
Coding considerations for sequential terminals . . 543

Print formatting 544
GOODNIGHT convention 544

Chapter 30. Using batch data interchange 545

Chapter 31. Terminal control: design for performance 549

Chapter 32. The 3270 family of terminals 551
The 3270 buffer. 551
The output datastream 551
3270 display fields. 553
Display characteristics 553
3270 field attributes 554
Extended attributes 556
Orders in the data stream 556
The start field order 557
The modify field order 558
The set buffer address order 558
The set attribute order 559
Outbound data stream example 559
Input from a 3270 terminal 562
Reading from a 3270 terminal 563
Inbound field format 564
Input data stream example 564
Unformatted mode 565

Part 9. Interval control and task control. 567

Chapter 33. Interval control 569
Expiration times 570

Chapter 34. Task control 573
Controlling sequence of access to resources . . . 574

Part 10. Storage protection and transaction isolation 577

Chapter 35. CICS storage protection and transaction isolation 579
Storage control 579
Storage protection 580
Storage categories 581
Transaction isolation 582
Defining the storage key for applications 583
System-wide storage areas 583
Task-lifetime storage 583
Program working storage specifically for exit and PLT programs. 584
Passing data by a COMMAREA 584
The GETMAIN commands 584
Selecting the execution and storage key 586
User-key applications. 586
CICS-key applications 587
Protection with transaction isolation. 589
MVS subspaces. 591

Part 11. Transient data and temporary storage 595

Chapter 36. Transient data control 597
Intrapartition transient data queues 597
Extrapartition queues. 598
Indirect queues. 599
Automatic transaction initiation (ATI) 599

Chapter 37. Temporary storage control 601

Part 12. CICS documents 603

Chapter 38. CICS documents and document templates 605
Symbols and symbol lists 606
Caching and refreshing of document templates . . 609
Code page conversion for documents 610

Chapter 39. Setting up document templates 613
Templates in a partitioned data set 613
Templates in z/OS UNIX System Services files . . 613
Templates in CICS files, temporary storage, or transient data 614
Templates in CICS programs 615
DFHDHTL - program template prolog and epilog macro 616
Templates in exit programs. 617
Communication area for templates in exit programs. 617
Using symbols in document templates 619
Embedded template commands 620

Chapter 40. Programming with documents and document templates . 623
Creating a document 623
Defining symbol values 624
Rules for specifying symbols and symbol lists . . 626
Adding more data to a document 629
Replacing data in a document 630
Retrieving, storing and reusing a document . . . 631
Deleting a document 634

Part 13. Named counter servers 635

Chapter 41. Named counter servers 637
The named counter fields 637
Named counter pools. 638
Named counter options table 638
Using the named counter EXEC interface 640
Using the named counter CALL interface 641
Application programming considerations . . . 641

Syntax 643
Return codes 651
Named counter recovery 653

Part 14. Printing and spool files 657

Chapter 42. Printing and spool files 659
Formatting for CICS printers 659
Requests for printed output 660
CICS 3270 printers 660
CICS 3270 printer options 661
PRINT option and print control bit 662
ERASE option 662
Line width options: L40, L64, L80, and HONEOM 662
NLEOM option. 663
FORMFEED 664
PRINTERCOMP option 664
Non-3270 CICS printers 665
SCS input 665
Using printers with CICS 666
Determining the characteristics of a CICS printer. 666
Using CICS printers 667
Using Non-CICS printers 670
Printing display screens 673
CICS interface to JES 674
Using the CICS interface to JES 676
Creating output spool files 677
Reading input spool files 678
Identifying spool files 679
Examples of SPOOL commands 681

Part 15. Basic Mapping Support (BMS) 685

Chapter 43. Basic mapping support 687
BMS support levels 687
A BMS output example 689
Creating the map 691
Defining map fields: DFHMDF 692
Defining the map: DFHMDI 694
Defining the map set: DFHMSD 694
Writing BMS macros 695
Assembling the map 697
Using complex fields 700
Block data 702
Support for non-3270 terminals 703
Device-dependent maps 705
Sending BMS mapped output 708
Acquiring and defining storage for the maps 709
Initializing the output map 710
Moving the variable data to the map 711
Setting the display characteristics. 711
Using the SEND MAP command 713
SEND MAP control options 714
Merging the symbolic and physical maps . . . 715
Building the output screen 716
Positioning the cursor 719
Sending invalid data and other errors 719
Output disposition options: TERMINAL, SET, and PAGING 720
Receiving mapped data 722
An input-output example 722
Programming mapped input 725
Using the RECEIVE MAP command. 725
Getting storage for mapped input 726
Formatted screen input 726
Using the attention identifier 728
Finding the cursor. 729
Processing the mapped input 729
Handling input errors 730
Sending mapped output after mapped input 732
MAPFAIL and other exception conditions . . . 733
Formatting other input 734
BMS logical messages 734
Building logical messages 735
The SEND PAGE command 736
RETAIN and RELEASE 736
The AUTOPAGE option 737
Terminal operator paging: the CSPG transaction 738
Logical message recovery 739
Cumulative output — the ACCUM option. . . . 739
Floating maps: how BMS places maps using ACCUM 739
Page breaks: BMS overflow processing 740
Map placement rules 741
Input from a composite screen. 743
Performance considerations. 744
Text output 745
The SEND TEXT command. 746
Text pages 746
Text lines 747
Header and trailer format 748
SEND TEXT MAPPED and SEND TEXT NOEDIT 749
Message routing 750
Message destinations 751
Route list format 753
Message delivery 755
Recoverable messages 756
Programming considerations with routing . . . 757
The MAPPINGDEV facility. 758
Sample assembler MAPPINGDEV application 760
Partition support 763
Uses for partitioned screens 764
Partition definition 765
Establishing partitioning. 766
Partition options for BMS SEND commands . . 767
Partition options for BMS RECEIVE commands 768
Partitions and logical messages 768
Attention identifiers and exception conditions 769
Terminal sharing 769
Support for special hardware 770
Logical device components 770
10/63 magnetic slot reader 772
Field selection features 773
Cursor and pen-detectable fields 774
Outboard formatting 775
BMS: design for performance 776

Page-building and routing operations 779

Part 16. Appendixes 783

Notices 785
Trademarks 786

Bibliography 787
CICS books for CICS Transaction Server for z/OS 787
CICSPlex SM books for CICS Transaction Server for z/OS 788
Other CICS publications 788
Other IBM publications 788

Accessibility 791

Index 793

What this manual is about

This manual documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM® CICS® Transaction Server Version 5 Release 1.

This manual 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 manual

This manual 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 manual

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 manual

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 manual 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.

Location of topics in the information center

The topics in this publication can also be found in the CICS information center. The information center uses content types to structure how the information is displayed.

The information center content types are generally task-oriented, for example; upgrading, configuring, and installing. Other content types include reference, overview and scenario or tutorial-based information. The following mapping shows the relationship between topics in this publication and the information center content types, with links to the external information center:

© Copyright IBM Corp. 1989, 2013

Table 1. Mapping of PDF topics to information center content types. This table lists the relationship between topics in the PDF and topics in the content types in the information center

Set of topics in this publication	Location in the information center
• Part 1, "Writing CICS applications," on page 1 and Part 2, "Programming languages and Language Environment," on page 9 • Part 3, "Translating, compiling, installing and testing application programs," on page 77 • All other parts	• Developing applications • Debugging • Deploying

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 manual

Guidance for usage of the CICS Front End Programming Interface is not discussed in this manual. 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 manual. See the *CICS Internet Guide* for this information.

Guidance for the use of object oriented programming languages and techniques is not included in this manual. 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*.

Changes in CICS Transaction Server for z/OS, Version 5 Release 1

For information about changes that have been made in this release, please refer to *What's New* in the information center, or the following publications:

- *CICS Transaction Server for z/OS What's New*
- *CICS Transaction Server for z/OS Upgrading from CICS TS Version 4.2*
- *CICS Transaction Server for z/OS Upgrading from CICS TS Version 3.2*
- *CICS Transaction Server for z/OS Upgrading from CICS TS Version 3.1*

Any technical changes that are made to the text after release are indicated by a vertical bar (|) to the left of each new or changed line of information.

© Copyright IBM Corp. 1989, 2013

Part 1. Writing CICS applications

© Copyright IBM Corp. 1989, 2013

Chapter 1. Writing CICS applications

An introduction to CICS applications and to the procedures for creating programs that run in CICS.

For reference information about the CICS Application Programming Interface, see CICS API commands in Reference -> Application development.

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. 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, CICS application programs are sometimes called *programs*, and the term *transaction* is used to imply the processing that the application programs do.

The term *transaction* is now used extensively in the IT industry to describe a *unit of recovery*, which in CICS is called 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 be aware of the difference in meaning when you use non-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.

The following terms are used throughout CICS documentation and in many commands:

Program

A program is used by CICS to process an entire transaction, or part of a transaction. Programs are stored in the program library. For more information about program resources and programs used in CICS bundles, see PROGRAM resources in Reference -> System definition.

Transaction

A transaction is an item of processing that is initiated by a single request. This request is usually from an end-user at a terminal, but can also be made from a web page, a remote workstation program, or an application in another CICS system, or triggered automatically at a predefined time.

© Copyright IBM Corp. 1989, 2013

For descriptions of different ways to run CICS transactions, see Internet, TCP/IP, and HTTP concepts in Product overview and Overview of CICS external interfaces in Product overview.

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 provides information to CICS about the work to be done; for example, which 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*, which in CICS is called 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 be aware of the difference in meaning when you use non-CICS documentation.

Task The term *task* has a specific meaning in CICS and CICS documentation. 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 specific set of data, usually on behalf of a specific user at a specific terminal. You can also consider a task as analogous to a thread. When the transaction completes, the task ends.

CICS programming

You write a CICS program in a similar way to 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 information describes the use of the CICS command level programming interface, **EXEC CICS**, that can be used in COBOL, C, C++, PL/I or assembler language programs. These commands are defined in detail in Application development reference. The following programming information is also available:

- Programming in Java with the JCICS class library is described in Java programming using JCICS in Developing applications.
- Programming in C++ with the CICS C++ classes is described in Using the CICS C++ OO foundation classes in Developing applications.
- For information about writing web applications to process HTTP requests and responses, see Developing HTTP applications in Developing applications.

For further guidance on language use with CICS, see Chapter 2, “Programming languages and Language Environment,” on page 11.

As well as CICS commands, you can use SQL statements, DLI requests, CPI statements, and the CICS Front End Programming Interface (FEPI) commands in your program. See the following information for the relevant details:

- To use SQL, see the *DB2 for z/OS reference: DB2 SQL* and *DB2 for z/OS Application Programming and SQL* in Information Management Software for z/OS Solutions Information Center.
- To use DL/I, see the *IMS: Application programming for EXEC DLI* and *IMS: Application programming design* in Information Management Software for z/OS Solutions Information Center.
- To use CPI, see z/VM Application Programming.
- To use FEPI, see FEPI application development in Developing applications.
- To use WebSphere MQ, see WebSphere MQ Library.

Related information:

System commands in Reference > System programming

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 ADDRESS and ASSIGN commands to access such information.

When you use the ADDRESS and ASSIGN commands, various fields can be read but should not be set or used in any other way. Do not use any of the CICS fields as arguments in CICS commands, because these fields might be altered by the EXEC interface modules.

You can use the INQUIRE, SET, and PERFORM commands for application programs to access information about CICS resources. These commands are known as system programming commands. 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 the Job Entry Subsystem (JES), are primarily for system programmers to use.

For reference information about CICS programming commands, see CICS API commands in Reference -> Application development.

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*.

Translation

Some older compilers (and assemblers) cannot process CICS commands directly. 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.

Most compilers 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, many of the translation tasks are done at compile time and you do not need to complete the additional translation step. For more information about the task in the translation step, see "The translation process" on page 81.

Testing for CICS

A program has two ways to determine whether it is running in a CICS environment: by using the C language iscics() function or by calling the DFH3QSS program.

About this task

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 can 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 into your own application. On return, register 15 addresses a result structure that consists of a halfword length (that includes itself) followed by a reserved halfword (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.

Environment

This function must be called in problem state.

CICS programming roadmap

Follow the steps in this roadmap to develop a CICS application that uses the EXEC CICS command level programming interface.

Procedure

1. Design your application, identifying the CICS resources and services you will use. See Chapter 18, “Application design,” on page 223 and “Design for performance” on page 252 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. Translate and compile your program. If you are using a compiler that incorporates “The integrated CICS translator” on page 79, you will only need to compile your program, and then install it in CICS, using the process described in “Program installation steps” on page 118. Otherwise, you will need to define translator options for your program, using the process described in “Using a CICS translator” on page 94, and then translate and compile your program, and install it in CICS, using the process described in “Program installation steps” on page 118.
4. Define your program and related transaction to CICS. Use PROGRAM resource definitions and TRANSACTION resource definitions as described in the*CICS Resource Definition Guide* .
5. Define and install any CICS resources that your program uses, such as files, queues or terminals.
6. Run your program. Enter the transaction identifier at a CICS terminal, or use any of the methods described in the *CICS External Interfaces Guide*, or Connecting CICS to other systems.

Identifying Programming Interfaces

The CICS information center contains some documentation that can be considered to be Programming Interfaces, and some documentation that cannot be considered to be a Programming Interface.

Programming Interfaces that allow the customer to write programs to obtain the services of CICS Transaction Server for z/OS, Version 5 Release 1 are included in the following sections of the information center:

- Developing applications
- Developing system programs
- Securing
- Developing for external interfaces
- Reference > Application development
- Reference > System programming
- Reference > Connectivity and standards

Information that is NOT intended to be used as a Programming Interface of CICS Transaction Server for z/OS, Version 5 Release 1, but that might be mis-construed as Programming Interfaces, is included in the following section of the information center:

- Troubleshooting and support
- Reference > Diagnostics

If you access the CICS documentation in PDF format, Programming Interfaces that allow the customer to write programs to obtain the services of CICS Transaction Server for z/OS, Version 5 Release 1 are included in the books in the following table:

Intended programming interface	Link to PDF
CICS Application Programming Guide	PDF
CICS Application Programming Reference	PDF
CICS Business Transaction Services	PDF
CICS Customization Guide	PDF
CICS C++ OO Class Libraries	PDF
CICS Debugging Tools Interfaces Reference	PDF
CICS Distributed Transaction Programming Guide	PDF
CICS External Interfaces Guide	PDF
CICS Front End Programming Interface User's Guide	PDF
CICS IMS Database Control Guide	PDF
CICS Intercommunication Guide	PDF
CICS Internet Guide	PDF
CICS RACF Security Guide	PDF
CICS Shared Data Tables Guide	PDF
CICSPlex System Manager Managing Workloads	PDF
CICSPlex System Manager Managing Resource Usage	PDF
CICSPlex System Manager Application Programming Guide	PDF
CICSPlex System Manager Application Programming Reference	PDF
Java Applications in CICS	PDF
JCICS Javadoc	N/A
Web services assistant Javadoc	N/A

Information that is NOT intended to be used as a Programming Interface of CICS Transaction Server for z/OS, Version 5 Release 1, but that might be mis-construed as Programming Interfaces, is included in the manuals in the following table:

Not intended as a programming interface	Link to PDF
CICS Data Areas	PDF
CICS Diagnosis Reference	PDF
CICS Problem Determination Guide	PDF
CICSPlex System Manager Problem Determination	PDF

Part 2. Programming languages and Language Environment

Information for programmers about Language Environment®, and language-specific guidance for programming in COBOL, C and C++, PL/I, and assembler language.

© Copyright IBM Corp. 1989, 2013

Chapter 2. Programming languages and Language Environment

Language Environment, supplied as an element of z/OS, provides a common set of runtime libraries. Language Environment allows you to use only one runtime environment for your applications, regardless of the programming language or system resource needs, because most system dependencies have been removed.

Before the introduction of Language Environment, each of the high-level languages (HLLs) provided a separate runtime environment. The runtime libraries provided by Language Environment replace the runtime libraries that were provided with older compilers such as VS COBOL II, OS PL/I, and C/370™. The common environment offers two significant advantages:

- You can mix all the languages supported by CICS in a single program.
- The same Language Environment callable services are available to all programs. For example:
  - 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 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.

CICS supports application programs compiled by a variety of compilers; for a list of compilers that are supported in this release of CICS Transaction Server for z/OS, see High-level language support.

Language Environment does not support AMODE(64) assembler language programs.

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 that are available to a CICS region. CICS and Language Environment also support programs compiled by some pre-Language Environment compilers, which do not conform with Language Environment. However, CICS does not support all the pre-Language Environment compilers that are supported by Language Environment.

Applications compiled and linked with pre-Language Environment compilers might run successfully using the runtime support provided by Language Environment. These applications might not require recompiling or re-link-editing. In some circumstances, you might need to adjust Language Environment runtime options so that the applications run correctly. See the *z/OS Language Environment Run-Time Application Migration Guide* and the *Compiler and Run-Time Migration Guide* for the language in use, for further information.

© Copyright IBM Corp. 1989, 2013

The runtime libraries provided with pre-Language Environment compilers are not supported. Do not include any Language libraries other than the Language Environment libraries in your CICS startup JCL.

When modifying existing application programs, or writing new programs, you must use a compiler supported by Language Environment. Your application programs must be link-edited using the Language Environment SCEELKED library, which means that the resulting application load module can run only under Language Environment.

In CICS you can also create Assembler MAIN programs that conform to Language Environment. For more information about Assembler programs, see Chapter 6, "Developing Assembler language applications," on page 61.

Language Environment callable services

Language Environment provides callable services, which can be accessed by programs running under CICS.

The callable services provided by Language Environment 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*.

Message and dump services

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

The usual destinations for Language Environment messages and dumps 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*.

Language Environment abend and condition handling

Language Environment abend handling depends on the use of CICS HANDLE ABEND. User-written condition handlers can be used when a CICS HANDLE ABEND is not active. Language Environment is not involved in the handling of CICS-defined exception conditions, or in the detection of attention identifiers (AIDs).

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, the action defined in the CICS HANDLE ABEND takes place. Language Environment condition handling does not gain control for any abends or program interrupts, and any user-written condition handlers that are established by CEEHDLR are ignored.
- When a CICS HANDLE ABEND is not active, Language Environment condition handling gains 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 occurs and the abend proceeds. For details of normal Language Environment condition handling, see the *z/OS Language Environment Programming Guide*.

User-written Language Environment condition handlers

You can use the Language Environment runtime option USRHDLR 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 create a user-written Language Environment condition handler (other than in COBOL), you can use most CICS commands, provided that 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, 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 an application to use a user-written Language Environment condition handler, the condition handler must be available at runtime (for example by using the STEPLIB concatenation or LPA). Define such condition handlers in the CICS system definition data set (CSD) for your CICS region, rather than using program autoinstall. This includes the sample user-written condition handler CEEWUCHA.

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

CICS condition and attention identifier (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.

Language Environment storage

Language Environment uses storage obtained from CICS for each run-unit. 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 the CICS system initialization parameter AUTODST=YES, CICS indicates to Language Environment that it is able to support dynamic storage tuning.

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 CICSDATAKEY or USERDATAKEY storage by using a CICS GETMAIN command. However, a program with an EXECKEY of USER cannot use CICSDATAKEY storage.

Mixing languages in Language Environment

You can use Language Environment to build an application that is composed of programs that are written in different high-level source languages and assembler language.

Assembler language subroutines called from a high level language (HLL) program are fairly straightforward and common. A subroutine that is called from one HLL, but written in another, needs much more careful consideration and involves interlanguage communication (ILC). Language Environment defines an ILC application as one that is built from two or more HLLs and, optionally, assembler language. For more information, see z/OS Language Environment Writing Interlanguage Communication Applications.

In Language Environment, if there is any ILC in a run unit under CICS, each compile unit must be compiled with a Language Environment-conforming compiler. CICS supports the following HLLs:

- C/C++
- COBOL
- PL/I

The following sections describe the conditions for each pair of HLLs. If your application contains only two HLLs, consult the appropriate section. If your application contains all three HLLs, consult the sections that correspond to each interface in 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 factors:

- Whether the language is C or C++.

- Which COBOL compiler is used and whether DLL is specified as a compiler option.
- Whether the call is static or dynamic.
- Whether the function that is invoked is in the module, or exported from a DLL.
- Whether the program is reentrant.
- What, if any, #pragma linkage statement is in your C program.
- Whether your C program exports functions or variables.
- What, if any, extern statement is in your C++ program.

For information about the effect of these factors, see z/OS Language Environment Writing Interlanguage Communication Applications.

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 C/C++ and PL/I 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, 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 C/C++ routines that were not processed by the CICS translator. This is because during the dynamic call, certain static fields created by the translator cannot be set correctly.

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, the calling routine must pass the EIB and the COMMAREA as the first two parameters on the CALL statement.

Assembler language

- You can make static or dynamic calls from any Language Environment-conforming HLL program to a Language Environment-conforming assembler language subroutine. Conversely, a Language Environment-conforming assembler language routine can make a static call to any Language Environment-conforming routine, and can dynamically load another routine, either assembler language or HLL, by using either of the Language Environment macros CEEFETCH or CEELOAD.
- You cannot delete (release) an ILC module that was loaded by using CEELOAD.
- You can use the CEERELES macro to release an ILC module that was fetched by using CEEFETCH.

- Use the language that fetched it to delete an assembler language routine. This can be done from C/C++, COBOL, and PL/I, only 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 language subroutine to a non-conforming assembler language subroutine. However, a non-conforming assembler language routine cannot make a static call to any Language Environment-conforming routine, and cannot fetch or load a conforming routine, because it cannot use the Language Environment macros.

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

C `#pragma linkage(,OS)`

C++ `extern "OS"`

DL/I

If you use 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.

Dynamic Link Libraries (DLLs)

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 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 following methods listed to set the Language Environment runtime options. For more information about the full order of precedence for Language Environment runtime options see *z/OS V1R11.0 Language Environment Programming Guide* SA22-7561-10. 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. The CEEDOPT CSECT built into CEECCICS contains the IBM Language Environment default runtime options. You can change these default runtime options by using the CEEWCOPT sample job located in SCEESAMP. This option is supported but using the CEEPRMxx parmlib member to specify runtime options is the preferred and easiest method.
2. The CEEPRMxx parmlib member provides support for the CEECOPT option group which is the preferred method for setting your default Language Environment runtime options for CICS.
3. 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.
4. The user replaceable program DFHAPXPO (applies to XPLINK programs only).
5. In the CEEUOPT CSECT, where user-supplied application program-level runtime options are located. This CSECT is linked with the application program itself.
6. 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.
7. In the Language Environment options specified in a debugging profile. For more information, see “Debugging profiles” on page 212.

In most installations, the first method in the previous list is unavailable to application programmers, and the second is often unavailable. However, application programmers can use method 4 or method 5. Choose one method only; do not attempt to use both method 4 and method 5. For information about generating a CEEUOPT CSECT to link with your application, see *z/OS Language Environment Customization*.

Both CEEDOPT and CEEROPT are able to set any option so that it cannot be overridden by a later specification.

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

Runtime options ignored under CICS

Under CICS many of the Language Environment runtime 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 (unless XPLINK or Java program)
- RTEREUS
- RTLS
- SIMVRD
- THREADHEAP
- VERSION

Determining which runtime options were used

If you want to know which Language Environment runtime 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.

Runtime options in child enclaves: performance considerations

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. These runtime options are independent of those options that existed in the creating enclave.

Frequent use of EXEC CICS LINK, and the individual setting of many runtime options, could affect performance. A static or dynamic call does not incur these overheads. If you must use CEEUOPT to specify options, specifying only those options that are different from the defaults improves 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 and CEECSTX user exits

These two Language Environment user exits can change some of the Language Environment runtime options.

- Setting the CEEAUE_A_OPTION return parameter of the CEEBXITA user exit can change options (apart from the LIBRARY, RTLS, STACK, and VERSION options).
- In the storage tuning user exit, CEECSTX, the options STACK, LIBSTACK, HEAP, ANYHEAP, and BELOWHEAP can be set.

The exits are called in the order in which they are listed above.

The storage tuning exit CEECSTX, 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.

Language Environment supplies a sample source version of CEEBXITA in the SCEESAMP library (it 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, because 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.

If you write your own version of CEEBXITA, you must write it in assembler language. 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

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

CICSVAR: CICS environment variable

CICS provides an environment variable called CICSVAR to allow the CONCURRENCY and API program attributes to be closely associated with the application program itself. You can specify this environment variable by using the Language Environment runtime option ENVAR.

CICSVAR can be used in a CEEDOPT CSECT to set an installation default, but it is most useful when it is set in a CEEUOPT CSECT that is link-edited with an individual program, or set by a #pragma statement in the source of a C or C++

program, or set by a PLIXOPT statement in a PL/I program. For example, when a program is coded to threadsafe standards, it can be defined as such without changing a PROGRAM resource definition, or it can 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 language, for PL/I, for COBOL, and for C and C++ programs (both those compiled with the XPLINK option, and those compiled without it), if the programs werecompiled using a Language Environment-conforming compiler. CICSVAR cannot be used for assembler language programs that are not Language Environment-conforming, or for Java programs.

The use of CICSVAR overrides the settings on a PROGRAM resource definition installed through the standard RDO interfaces, or through program autoinstall. Before the program is run for the first time, an INQUIRE PROGRAM command shows the keyword settings from the program definition. When the application has run once, an INQUIRE PROGRAM command shows the settings with any CICSVAR overrides applied.

Valid values for CICSVAR are QUASIRENT, THREADSAFE, REQUIRED, and OPENAPI.

CICSVAR=QUASIRENT
: Results in a program with the attributes CONCURRENCY(QUASIRENT) and APIST(CICSAPI).

CICSVAR=THREADSAFE
: Results in a program with the attributes CONCURRENCY(THREADSAFE) and APIST(CICSAPI).

CICSVAR=REQUIRED
: Results in a program with the attributes CONCURRENCY(REQUIRED) and APIST(CICSAPI).

CICSVAR=OPENAPI
: Results in a program with the attributes CONCURRENCY(REQUIRED) and APIST(OPENAPI).

The following example shows the Language Environment runtime option ENVAR coded in a CEEUOPT CSECT:

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

This code can be assembled and link-edited into a load module, and then the CEEUOPT load module can be link-edited together with any language program supported by Language Environment.

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=THREADSAFE))
```

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=THREADSAFE)');
```

CEEBINT 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. Language Environment calls this program the High Level Language (HLL) user exit.

Language Environment provides a module containing this program in the SCEELKED library (it 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 language. 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 15 apply.

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

Chapter 3. Developing COBOL applications

Use this information to help you code, translate, and compile COBOL programs that you want to use as CICS application programs.

High-level language support in What's new lists the COBOL compilers that are supported by CICS Transaction Server for z/OS, Version 5 Release 1, and their service status on z/OS.

All references to COBOL in CICS Transaction Server for z/OS, Version 5 Release 1 documentation imply the use of a supported Language Environment-conforming compiler such as Enterprise COBOL for z/OS, unless stated otherwise. The only COBOL compiler that has runtime support in CICS Transaction Server for z/OS, Version 5 Release 1, but is not Language Environment-conforming, is the VS COBOL II compiler.

See the Enterprise COBOL for z/OS Compiler and Runtime Migration Guide for information about migrating between COBOL compilers.

Support for VS COBOL II

In CICS Transaction Server for z/OS, Version 5 Release 1, applications compiled with a VS COBOL II compiler run using the Language Environment runtime library routines. The runtime library provided with VS COBOL II is not supported.

"VS COBOL II programs" on page 28 lists some restrictions and considerations associated with programs compiled with the VS COBOL II compiler.

The VS COBOL II compiler can adjust Language Environment runtime options to allow these applications to run correctly. For more information, see the Enterprise COBOL for z/OS Compiler and Runtime Migration Guide.

Support for OO COBOL

In CICS Transaction Server for z/OS, Version 5 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 run in CICS Transaction Server for z/OS, Version 5 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, rewrite your OO COBOL into procedural (non-OO) COBOL in order to use the Enterprise COBOL compiler. 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.

© Copyright IBM Corp. 1989, 2013

Working storage

With compiler option DATA(24), the working storage is allocated below the 16 MB line. With compiler option DATA(31), the working storage is allocated above the 16 MB line.

COBOL programming restrictions and requirements

Some restrictions and requirements apply to a COBOL program that is used as a CICS application program.

By default, the CICS translator and the COBOL compiler do not detect the use of COBOL words that are affected by the following restrictions. The use of a restricted word in a CICS environment might 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 the IGYCCICS table, and the compiler flags COBOL words that are not supported under CICS with an error message. The COBOL words that are normally restricted by the default IBM-supplied reserved-word table are also flagged. For a current listing of the words that are restricted and are in the IGYCCICS table, see Enterprise COBOL for z/OS Programming Guide.

Functions and statements that cannot be used

- You cannot use entry points in COBOL in CICS.
- 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”.
- Restrictions apply to the use of the SORT statement; see Enterprise COBOL for z/OS Programming Guide. Do not use MERGE.
- Do not use the following:
  - USE declaratives.
  - ENVIRONMENT DIVISION and FILE SECTION entries associated with data management, because CICS handles data management. You can use these entries when they are associated with the limited SORT facility, listed previously.
  - User-specified parameters to the main program.

Coding requirements

- When a debugging line is to be used as a comment, it must not contain any unmatched quotation marks.
- Statements that produce variable-length areas, such as OCCURS DEPENDING ON, should be used with caution within the WORKING-STORAGE SECTION.
- Do not use EXEC CICS commands in a Declaratives Section.

- 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.
- 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 16 MB line.
  - When the compiler option DATA(31) is used, the limit is 128 MB.

  For storage accounting and save areas, 80 bytes are required; you must include this in the limits.
- If the DLI option is specified and an ENTRY statement immediately follows the PROCEDURE DIVISION header in an existing program, change the PROGRAM-ID name to the ENTRY statement literal and delete the ENTRY statement before calling the program in CICS.
- 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.

Language Environment coding requirements

If you are running CICS applications written in COBOL under Language Environment for the first time, you might 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. For information about customizing Language Environment runtime options, see z/OS Language Environment Programming Reference.

31-bit addressing

For a COBOL program running above the 16 MB line, the following 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.

64-bit addressing

64-bit addressing mode is not supported for COBOL programs.

Compiler options

- 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 following compiler options have no effect in a CICS environment:
  - ADV
  - AWO
  - EXPORTALL
  - FASTSRT
  - NAME
  - OOCOBOL
  - OUTDD
  - THREAD
- The use of the TEST(SYM,NOSEPARATE) compiler option results in a very large increase in program size. Therefore, short-on-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 use this option only if you have no control over binary data items (such as those created by a code generator). (TRUNC(STD) is the default.)

  Note that if your application uses fields in the EIB, the DFHEIBLK copybook defines fields such as EIBCALEN as PICTURE S9(4) USAGE COMPUTATIONAL. Using the TRUNC(OPT) compiler option with the DFHEIBLK copybook can result in the truncation of values greater than 9999 in binary fields. To avoid truncation problems, you are recommended to use the integrated translator which uses an updated version of the DFHEIBLK copybook. The version of DFHEIBLK used by the integrated translator defines all fields that would be affected by the TRUNC(OPT) or TRUNC(BIN) compile option as USAGE COMP-5.

  For more information about the TRUNC option, see the Enterprise COBOL for z/OS Customization Guide.
- The use of the RMODE(24) compiler option means that the program always resides below the 16 MB 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.

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.

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 17.

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.
- If you call an assembler routine and need to suspend the current handles, and then reinstate them, the assembler routine must request the push and pop handles. The Language Environment does not do that automatically when a COBOL program calls an assembler routine.

Using the DL/I CALL interface

If you have COBOL 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 28 illustrates the whole of the this 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

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 some restrictions and considerations are associated with its use.

For detailed information about upgrading 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, it is advisable to use the DFHELII stub, and it is essential to use the DFHELII stub in a mixed language environment. DFHECI must 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 functions as follows:

- 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.)

Ensure that the sample user condition handler, CEEWUCHA, is available at runtime (for example by using the STEPLIB concatenation or LPA). Define the condition handler in the CICS system definition data set (CSD) for your CICS region, rather than using program autoinstall.

Using based addressing with COBOL

COBOL provides a simple method of obtaining addressability to CICS data areas defined in the LINKAGE SECTION using pointer variables and the ADDRESS special register.

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.

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.

Figure 2 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

Calling subprograms from COBOL programs

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. In COBOL, 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 programs in any language supported by CICS, statically or dynamically. LINK or XCTL are not required for inter-language communication, unless you want to use CICS functions such as COMMAREA. See "Mixing languages in Language Environment" on page 15 for more information about inter-language communication.

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.

Flow of control between programs and subprograms

There are a number of possible flows between COBOL main programs and subprograms.

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.

Each LINK command creates a new **CICS application 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 32 shows logical levels and the effect of RETURN commands and CALL statements in linked and called programs.

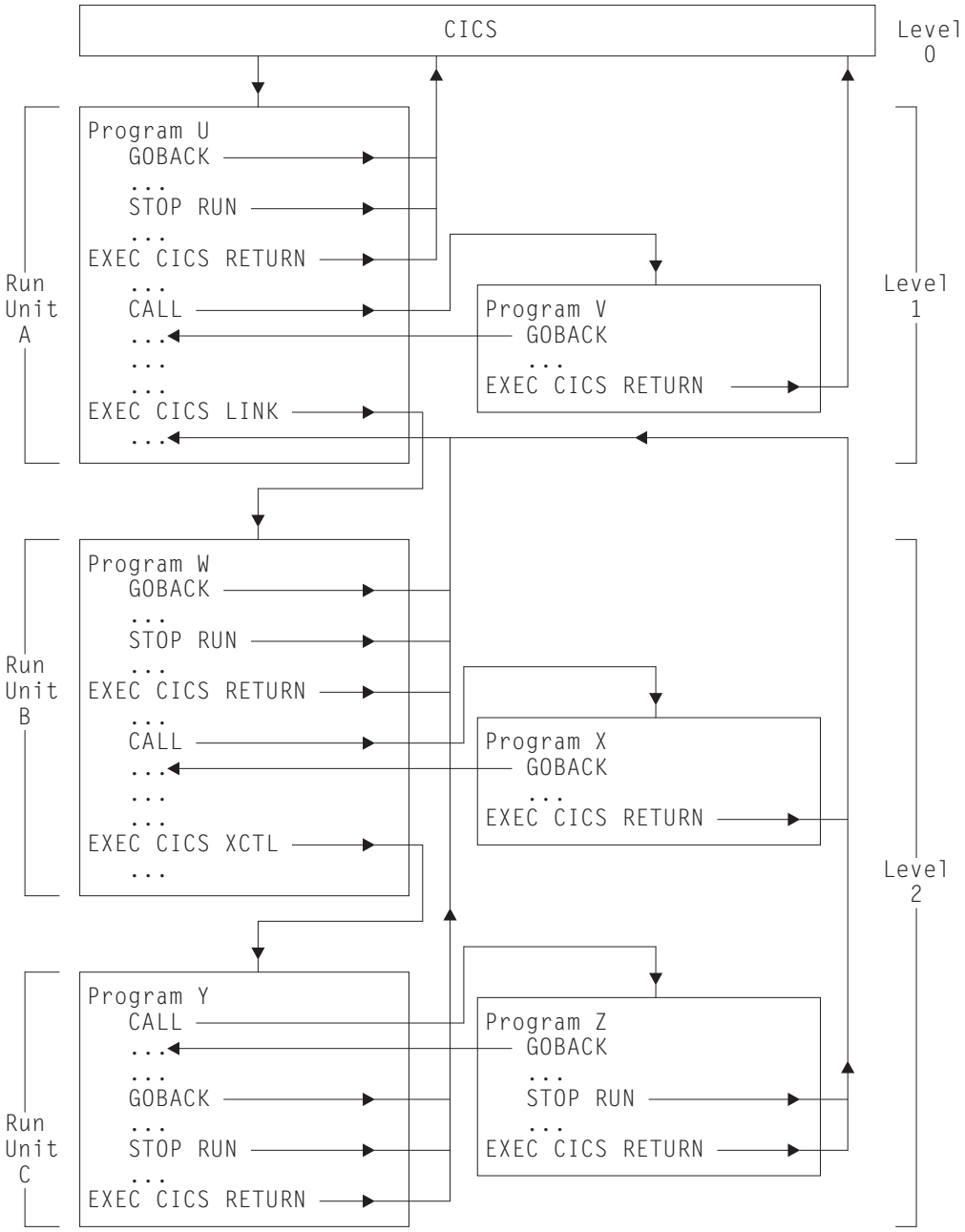


Figure 3. Flow of control between COBOL programs, run units, and CICS

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 301 for more information about program logical levels.

Rules for calling subprograms

These rules describe the requirements and behavior of subprograms called or linked from a COBOL program. The rules which apply depend on how control is transferred to the subprogram, whether by an **EXEC CICS LINK** command, a static COBOL call, or a dynamic COBOL call.

Location of subprogram

EXEC CICS LINK
The subprogram can be remote.

Static or dynamic COBOL call
The subprogram must be local.

Translation

If a compiler with an integrated translator is used, translation is not required.

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

Static or 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

EXEC CICS 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 require an entry in the CSD.

EXEC CICS 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).

Recursive calls in COBOL

If program A calls program B and program B attempts to call program A, Language Environment issues message IGZ0064S to CEEMSG and an abend (4038).

If program A and program B have the RECURSIVE keyword on the PROGRAM-ID, recursive calls are allowed.

Passing parameters to a 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.

EXEC CICS 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 16 MB line, CICS copies it to below the 16 MB line, and recopies it on return.

Static COBOL call
The CALL statement can 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.

Dynamic COBOL call
The CALL statement can 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) .

Return from a subprogram

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

Static or 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.

Storage

EXEC CICS 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).

On each entry to the linked subprogram, a new initialized copy of its LOCAL-STORAGE section is provided.

Static or 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 might give improved results.

On every entry to the called subprogram in a CICS logical level, a new initialized copy of its LOCAL-STORAGE SECTION is provided.

CICS condition, AID, and abend handling

EXEC CICS 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 or dynamic COBOL call

If the dynamically called COBOL program abends, CICS abend handling is not invoked, and you might get a COBOL abend code (1013).

- If the dynamically called COBOL program abends, 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 dynamically called COBOL program abends, with CBLPSHPOP OFF, the condition, AID, and abend handling for the calling program remain in effect.

COBOL2 and COBOL3 translator options

In CICS Transaction Server for z/OS, Version 5 Release 1, you can choose between the COBOL2 and COBOL3 CICS translator options for COBOL programs.

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 5 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 upgrading 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 programs 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.

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 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*.

For general information about translating your program and preparing it for execution, see Chapter 7, “Translation and compilation,” on page 79.

CICS translator actions for COBOL programs

These notes 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.

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*

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 clause is supported. A variable defined with the GLOBAL clause in a top-level program (see "Nested COBOL programs" on page 41) 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.

Batch compilation for COBOL programs

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 117.

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 40 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 might find it convenient to place them at the end of the module, immediately before each NAME statement. Figure 5 on page 41 shows the output from Figure 4 on page 40 after editing in this way.

For batch compilation you must vary the procedure described in Chapter 9, “Installing application programs,” on page 117. 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 on page 41. 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 CICSTS51.CICS.SDFHLOAD. The INCLUDE statement for the stub refers to this library by the DD name. In Figure 5 on page 41, it is assumed you have used the DD name SYSLIB (or concatenated this library to SYSLIB). The suggested statement is:

   ```
   //SYSLIB DD DSN=CICSTS51.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 COBOL 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*.

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 must be submitted together to the translator.

Comments in nested programs

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

If you are using a separate translator

If you are using a separate translator, and not using the integrated CICS translator, for nested programs that contain EXEC CICS commands, you need to explicitly code EIB and COMMAREA on the USING phrases on CALL and on the PROCEDURE DIVISION, as described in this section.

If you are using the integrated CICS translator, this action is **not** 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 is not required.

If you are using a separate translator:

1. 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 ...
   ```

2. 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 ...
   ```

3. 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.

4. If it is not necessary to insert DFHEIBLK and DFHCOMMAREA in the PROCEDURE DIVISION of a nested program for any of the reasons previously listed, 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 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.

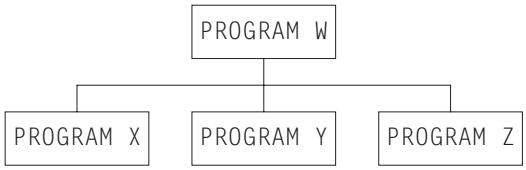


Figure 6. Nested program example—nesting structure

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, so if you are using a separate translator:
  - 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 if you are using a separate translator, 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 44 illustrates these points.

```
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

Chapter 4. Programming in C and C++

Use this information to help you code, translate, and compile C and C++ programs that you want to use as CICS application programs.

High-level language support lists the C and C++ compilers that are supported by CICS Transaction Server for z/OS, Version 5 Release 1, and their service status on z/OS. All references to C and C++ in CICS Transaction Server for z/OS, Version 5 Release 1 documentation imply the use of a supported Language Environment-conforming compiler, unless stated otherwise. All **EXEC CICS** commands available in COBOL, PL/I, and assembler language applications are also supported in C and C++ applications, except those commands related to nonstructured exception handling.

C++ applications can also use the CICS C++ OO classes to access CICS services, instead of the **EXEC CICS** interface. See Using the CICS C++ OO foundation classes in Developing applications for more information about this interface. 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.

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 16 MB line, is controlled by the ANYWHERE and BELOW options on the stack and heap runtime options. The default is that both the stack and heap are located above the 16 MB line.

Sample programs

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

Table 2. 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 can be installed using the command:

```
CEDA INSTALL GROUP(DFH$DFLA)
```

© Copyright IBM Corp. 1989, 2013

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

- DFH$DFIL: FILEA record descriptor
- DFH$DL86: L860 record descriptor

FLOAT compiler option

For z/OS V1.11 XL C (or C++) or later, specify either the FLOAT(NOAFP) compiler option, or the FLOAT(AFP(VOLATILE)) compiler option.

- If your program makes little or no use of floating point, specify the FLOAT(NOAFP) option. The program uses only the traditional four 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) option or the FLOAT(NOVOLATILE) option. The program can use all 16 floating point registers and CICS preserves the floating point registers used by the program.
- If you specify the FLOAT(AFP(VOLATILE)) option, CICS, C, and C++ preserve the floating point registers. Extra code is generated and can therefore degrade performance.

C and C++ programming restrictions and requirements

Some restrictions and requirements apply to a C or C++ program that is used as a CICS application program.

Functions and commands that cannot be used

The following EXEC CICS commands, which are related to nonstructured exception handling, are not supported for C and C++ applications:

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

The translator diagnoses use of these commands. **HANDLE ABEND PROGRAM** commands are allowed.

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

CICS does not support extended precision floating point.

C++ does not support packed decimal data. The application has access to packed decimal data using the character string data type. No C++ standard library functions are available to perform arithmetic on this data, but you can write your own. When using CICS commands that have options to specify time (for example, the **DELAY** or **POST** commands), it is advisable to use the HOURS, MINUTES, and SECONDS options. You can define times by using the TIME or INTERVAL options, which are packed decimal data types, if you provide functions to handle them in your application.

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

Native C or C++ file operations operate only on files that are opened with *type=memory* specified. I/O to CICS-supported access methods must use the CICS API.

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 might work but can impact the performance or execution of CICS.

- CDUMP
- CSNAP
- CTEST
- CTRACE
- CLOCK (The clock() function returns a value (time_t) of -1.)
- CTDLI
- SVC99
- SYSTEM
- SETLOCALE

Coding requirements

- You can enter all CICS keywords in mixed case, except for CICS keywords on #pragma directives, which must be in upper case only.
- 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.
- Do not use field names that might be acceptable to the assembler, but that cause the C or C++ compiler to abend. These names include $, #, and @.
- C++ uses '//' for single line comments. Do not put such a comment in the middle of an EXEC CICS command. For example, the following code is not valid:

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

  The following code 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);
```

Condition handling

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

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 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 ADDRESS EIB to obtain the address of the EIB. See "Accessing the EIB from C and C++" on page 51 for more information.

LENGTH

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.

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.

Addressing mode

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

Consequently, when you pass parameters to a program that is produced by the Cross-System Product (CSP) interactive application generator, you must do one of the following:

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

64-bit addressing mode (AMODE(64)) is not supported for C and C++ language programs.

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.

Data declarations

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

- Execution interface block definitions (EIB). 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++.

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

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.

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

The iscics() function can be useful 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 function returns a non-zero value if your program is currently running under CICS, or zero if it is not. This function is an extension to the C library.

String handling functions

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.

argc and argv arguments

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.

Passing arguments in C and 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 3 shows the rules that apply when passing values as arguments in EXEC CICS commands.

Table 3. 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 3. 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 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 from C and C++

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.

You must code an ADDRESS EIB statement at the beginning of each application if you want access to the EIB, or if you are using a command that includes the RESP or RESP2 option.

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.

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”.

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.

XPLink and C and C++ programming

CICS provides support for C and C++ programs compiled with the XPLINK option. All programs using CICS XPLink support must be reentrant and threadsafe.

Extra Performance Linkage, normally 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.

To use XPLink, your C or C++ application code must be **reentrant** 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.

If you plan to compile C and C++ programs for the CICS environment with the XPLINK option, 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 the note in "Defining runtime options for Language Environment" on page 17 for details).

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

XPLink uses X8 and X9 mode 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.

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.

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, and 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.

Global user exits and XPLink

The XPCFTCH and XPCTA exits are affected by the use of the XPLINK option. CICS disregards any attempt by XPCFTCH to modify the entry point, and any attempt by XPCTA to define a resume address. Other global user exits are unaffected by XPLink support.

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.

These activities are ignored because the batch Language Environment runtime used for XPLink programs does not give control to CICS when a program abends, but goes through its own abend handling. When control reaches CICS, the Language Environment enclave has terminated, so CICS is unable to honor an entry point address or a resume address.

If you have an application program that carries out these activities, you must find other ways to manage these requirements, or conclude that the program is not a suitable candidate for XPLINK optimization. One possible solution is to write a Language Environment abnormal termination exit, as described in "Customizing user exits" in the *z/OS Language Environment Customization* manual.

Chapter 5. Developing PL/I applications

Use this information to help you code, translate, and compile PL/I programs that you want to use as CICS application programs.

High-level language support lists the PL/I Compilers that are supported by CICS Transaction Server for z/OS, Version 5 Release 1, and their service status on z/OS.

All references to PL/I in CICS Transaction Server for z/OS, Version 5 Release 1 documentation imply the use of a supported Language Environment-conforming compiler, unless stated otherwise.

OPTIONS(MAIN) specification

If you specify the OPTIONS(MAIN) option in a PL/I application program, that program can be the first program of a transaction, or control can be passed to it with a LINK or XCTL command.

In PL/I application programs where the OPTIONS(MAIN) option 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.

FLOAT compiler option

For Enterprise PL/I, specifying the FLOAT option controls the use of the additional floating point registers.

- If your program makes little or no use of floating point, specify the FLOAT(NOAFP) option. The program uses the traditional four 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) option or the FLOAT(NOVOLATILE) option. The program can use all 16 floating point registers, and CICS preserves the floating point registers used by the program.
- If you specify the FLOAT(AFP(VOLATILE)) option, both CICS and PL/I preserve the floating point registers. Extra code is generated and performance can be affected as a result.

PL/I programming restrictions and requirements

Some restrictions and requirements apply to a PL/I program that is used as a CICS application program.

Functions and statements that cannot be used

- You cannot use the following multitasking built-in functions:

  ```
  COMPLETION
  PRIORITY
  STATUS
  ```

- You cannot use the following multitasking options:

  ```
  EVENT
  PRIORITY
  ```

© Copyright IBM Corp. 1989, 2013

```
TASK
```

- You should not use the following PL/I statements:

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

  The FETCH and RELEASE statements are supported. EXEC CICS commands are provided 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).

Coding requirements

- If you declare a variable with the STATIC attribute and EXTERNAL attribute, you must also include the INITIAL attribute. If you do not, such a declaration generates a common CSECT that cannot be handled by CICS.
- Do not define variables or structures with variable names that are the same as variable names generated by the translator. These begin with DFH. Use care with the LIKE keyword to avoid implicitly generating such variable names.
- All PROCEDURE statements must be in uppercase, except for the PROCEDURE name, which can be in lowercase.
- The suboptions of the XOPTS option of the *PROCESS statement must be in uppercase.
- You cannot use the PL/I 48-character set option in EXEC CICS statements.
- 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 following form:

  ```
  CSTG(SUBSTR(..,..,..))
  ```

  This call is rejected by the compiler.

64-bit addressing

64-bit addressing mode is not supported for PL/I programs.

Language Environment coding requirements for PL/I applications

All PL/I programs are executed under the runtime support provided by Language Environment. There are some additional coding requirements compared to pre-Language Environment PL/I programs.

Language Environment runtime options, if needed, can be specified in a **plixopt** character string. See "Defining runtime options for Language Environment" on page 17 and the *z/OS Language Environment Programming Reference* for information about customizing runtime options.

If you are converting a PL/I program that was previously compiled with a non-Language Environment-conforming compiler, you must ensure that NOSTAE and NOSPIE are not specified in a **plixopt** string, because either of these will cause Language Environment to set TRAP (OFF). TRAP (ON) must be in effect for applications to run successfully.

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. 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. This input file replaces 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 Compiler and Runtime Migration Guide*.

Communicating between conforming and non-conforming PL/I routines

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

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.

Abend handling

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/I abend 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*.

Ensure that the sample user condition handler, CEEWUCHA, is available at runtime (for example by using the STEPLIB concatenation or LPA). Define the condition handler in the CICS system definition data set (CSD) for your CICS region, rather than using program autoinstall.

Fetched PL/I routines

To enable a PL/I procedure to be fetched, code the option FETCHABLE in the OPTIONS on the PROCEDURE statement.

The FETCHABLE option indicates that the procedure can only be invoked dynamically. An OPTIONS(MAIN) procedure cannot be fetched; FETCHABLE and MAIN are mutually exclusive options.

For Enterprise PL/I programs that are compiled with OPTIONS(FETCHABLE), the binder ENTRY statement must be the name of the PROCEDURE.

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.

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

There were some restrictions on the PL/I for MVS & VM statements that could be used in a fetched procedure. These restrictions are described in *PL/I MVS & VM Language Reference*. Many of the restrictions were removed with VisualAge® PL/I. See the *VisualAge PL/I Compiler and Runtime Migration Guide*.

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.

Chapter 6. Developing Assembler language applications

Use this information to help you code assembler language programs that you want to use as CICS application programs.

Working storage

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

Sample programs

A set of sample application programs is provided to show how **EXEC CICS** commands can be used in a program written in assembler language. These programs are AMODE(64) and use relative addressing, except for DFH$AREP. DFH$AREP uses relative addressing but is AMODE(31) because it demonstrates the use of the HANDLE CONDITION LABEL command.

Table 4. Sample programs

Sample program	Map set	Map source	Transaction ID
DFH$AMNU **Operator instruction (3270)**	DFH$AGA	DFH$AMA	AMNU
DFH$AALL **Update (3270)**	DFH$AGB	DFH$AMB	AADD, AINQ, AUPD
DFH$ABRW **Browse (3270)**	DFH$AGC	DFH$AMC	ABRW
DFH$AREN **Order entry (3270)**	DFH$AGK	DFH$AMK	AORD
DFH$ACOM **Order entry queue print (3270)**	DFH$AGL	DFH$AML	AORQ
DFH$AREP **Report (3270)**	DFH$AGD	DFH$AMD	AREP

The transaction and program definitions are provided in group DFH$AFLA in the CSD and can be installed using the following command:

```
CEDA INSTALL GROUP(DFH$AFLA)
```

The following record description files are provided:

- DFH$AFIL: FILEA record descriptor
- DFH$AL86: L860 record descriptor

Assembler language programming restrictions and requirements

Some restrictions and requirements apply to an assembler language program that is used as a CICS application program.

LEASM option

The following restrictions apply to an assembler language program that is translated with the LEASM option:

- Register 2 cannot be used as a code base register.

© Copyright IBM Corp. 1989, 2013

- Register 12 is reserved by Language Environment (LE) 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 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.
- AMODE(64) programs are not supported.

LENGTH option in EXEC CICS commands

When you specify a LENGTH option for a CICS command, ensure that the way you use in assembler language specifies a valid halfword length. Do not specify a zero length, or a variable that the CICS translator cannot recognize, because this might cause a storage violation or a program check. For more information, see Assembler language argument values in Developing system programs.

The following example of a LINK command specifies the length correctly and passes the address of the variable COMMAL to the command processor:

```
EXEC CICS LINK PROGRAM('PROG2') COMMAREA(COMMA) LENGTH(COMMAL)
...
COMMA DC CL20'This is the COMMAREA'
COMMAL DC H'20'
```

The following example also specifies the length correctly:

```
EXEC CICS LINK PROGRAM('PROG2') COMMAREA(COMMA) LENGTH(=AL2(COMMAL))
...
COMMA DC CL20'This is the COMMAREA'
COMMAL EQU *-COMMA
```

The following example is incorrect because the CICS translator cannot know the type of the variable COMMAL, and passes the value of COMMAL as the address of the halfword field that contains the length. This could provide a random length value, or a program check if the storage at that address is not available.

```
EXEC CICS LINK PROGRAM('PROG2') COMMAREA(COMMA) LENGTH(COMMAL)
...
COMMA DC CL20'This is the COMMAREA'
COMMAL EQU *-COMMA
```

31-bit addressing

The following restrictions apply to an assembler language application program that runs in 31-bit addressing mode:

- The interval control command WAIT EVENT is not supported when the associated event control block (ECB) resides above the 16 MB line. You can use the task control command WAIT EXTERNAL with the ECBLIST option instead.
- 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 306.
- CICS does not allow the use of HANDLE ABEND LABEL in assembler language programs that do not use the DFHEIENT and DFHEIRET macros. Assembler language programs that use the Language Environment stub CEESTART should either use HANDLE ABEND PROGRAM or a Language Environment service such as CEEHDLR. See “Language Environment abend and condition handling” on page 13 for information about CEEHDLR.

The following restriction applies to an AMODE(24) or AMODE(31) assembler language application program that uses 64-bit registers to use 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.

64-bit addressing

CICS supports the program execution of non-Language Environment AMODE(64) assembler language CICS applications. Your program must use relative addressing. See "Developing AMODE(64) assembler language programs" on page 75. All CICS API commands, except for those listed later in this section, are supported. For information about the CICS API, see CICS API commands in Reference -> Application development.

The following APIs are not supported for AMODE(64) programs:

- CICSPlex® SM application programming interface (API)
- CICS API commands for use in APPC basic conversations (GDS commands)
- Front End Programming Interface (FEPI)
- Common Programming Interface (CPI) Communications API
- DL/I requests

The following interfaces are not supported for AMODE(64) programs:

- CICS DB2 interface
- CICS-MQ bridge external interface

The following CICS API commands are provided for use with AMODE(64) programs:

- FREEMAIN64
- GETMAIN64
- GET64 CONTAINER
- PUT64 CONTAINER

The following CICS API commands, which are related to nonstructured exception handling, are not supported for AMODE(64) assembler language programs:

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

The translator detects use of these commands.

AMODE(64) assembler language programs cannot be invoked by a COBOL, C, C++, or PL/I CALL statement, or by an assembler language CALL macro. However, assembler language application programs can be invoked by COBOL, C, C++, PL/I, or assembler language application programs by using LINK or XCTL commands.

For AMODE(64) assembler language programs, use of the NOHANDLE condition is implied on all EXEC CICS commands. This means that no action will be taken for any condition that results from the execution of that command. You can use the RESP option with any command to test whether any condition was raised during execution of that command. See RESP and RESP2 options in Reference -> Application development and NOHANDLE option in Reference -> Application development.

For AMODE(64) assembler language programs, CICS uses the whole of the 64-bit register. Ensure that all of the register is valid.

A COMMAREA cannot be in 64-bit storage. For more information the COMMAREA, see "COMMAREA" on page 302.

AMODE 64 task-related user exits (TRUEs) are not supported, and AMODE(64) applications cannot invoke a TRUE.

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

Access registers

The following restrictions apply to an assembler language application program that uses access registers to use the extended addressability of z/Architecture® systems:

- 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 z/OS MVS Programming: Extended Addressability Guide.

BAKR instructions (branch and stack)

When using BAKR instructions (branch and stack) to provide linkage between assembler language programs, ensure 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), other tasks might be dispatched and issue further BAKR / PR calls. These calls modify the linkage-stack and result in the wrong environment being restored when the original task issues its PR instruction.

Instructions that cannot be used

You cannot use the following instructions in an assembler language program that is used as a CICS application program:

COM Identify blank common control section.

ICTL Input format control.

OPSYN
Equate operation code.

Coding the EXEC CICS assembler interface

For AMODE(24) and AMODE(31) assembler language programs, you can optionally specify your own DFHEIENT macro. To declare that an assembler language programs is AMODE(64), you use the z/OS Assembler SYSSTATE macro. For AMODE(64) assembler language programs, you must specify the DFHEIENT macro, and you can optionally specify your own DFHEIRET macro.

AMODE(24) and AMODE(31) assembler language programs

DFHEIENT macro

For AMODE(24) and AMODE(31) programs, the values provided by the DFHEIENT macro that the translator inserts automatically might be inadequate for application programs that produce a translated output greater than 4095 bytes. In this situation, you can provide your own version of the DFHEIENT macro. To prevent the translator from automatically inserting its version of the DFHEIENT macro, specify the NOPROLOG translator option.

For example, by default, the translator sets up only one base register (register 3), which might cause addressability problems for your program. You can provide your own DFHEIENT macro with the CODEREG operand so that you can specify more than one base register. If you code your own version of the DFHEIENT macro with the same label as the CSECT statement, it can replace the CSECT statement in your source program. If you code the DFHEIENT macro without a label, it must immediately follow the CSECT statement.

You can use the following operands to specify base registers:

- CODEREG - base registers
- DATAREG - dynamic-storage registers
- EIBREG - register to address the EIB

For example, the following simple assembler language application program uses the BMS command SEND MAP to send a map to a terminal.

```
INSTRUCT CSECT
         EXEC CICS SEND MAP('DFH$AGA') MAPONLY ERASE
         END
```

The following example code increases the number of base and data registers for this program:

```
INSTRUCT DFHEIENT CODEREG=(2,3,4),
         DATAREG=(13,5),
         EIBREG=6
         EXEC CICS SEND
         MAP('DFH$AGA')
         MAPONLY ERASE
         END
```

The symbolic register DFHEIPLR is equated to the first DATAREG either explicitly specified or obtained by default. The DFHECALL macro ensures that register 13 addresses the save area that DFHEISTG defined in dynamic storage, so it is advisable to use register 13 as your first data dynamic-storage register. If you do not, the code generated by DFHECALL adds extra instructions to manipulate register 13.

DFHEIPLR is assumed by the expansion of a CICS command to contain the value set up by DFHEIENT. You should either dedicate this register or ensure that it is restored before each CICS command.

You can also use the DFHEIENT macro to specify that you want to use relative addressing instructions in your program. When you use relative addressing, you do not need to use any base registers to address your program instructions, but you must use at least one register to address static data in your program. Specify the following operands on the DFHEIENT macro:

- CODEREG=0 to specify that no registers are to be used to address program instructions.
- STATREG to specify one or more registers to address the static data area in your program.
- STATIC to specify the address of the start of the static data in your program.

If you use relative addressing, use the IEABRCX DEFINE macro (provided by z/OS) to redefine the assembler mnemonics for branch instructions to use relative branch instructions. Also, ensure that any LTORG statements, and instructions that are the target of EXECUTE instructions, appear after the label specified in the STATIC operand. For example:

```
         IEABRCX DEFINE                Define relative branch mnemonics
RELATIVE DFHEIENT CODEREG=0,STATREG=(8,9),STATIC=MYSTATIC
         ....
         EX    R2,VARMOVE              Execute instruction in static area
         ....

MYSTATIC DS    0D                      Static data area
MYCONST  DC    C'constant'             Static data value
VARMOVE  MVC   WORKA(0),WORKB          Executed instruction
         LTORG ,                       Literal pool
```

For more information about the IEABRCX macro, see IEABRCX — Relative branch macro extension in z/OS MVS Programming: Assembler Services Reference, Volume 2.

Assembler language programs that are translated with the DLI option have a DLI initialization call inserted after each CSECT statement. Assembler language programs that are larger than 4095 bytes and that do not use the CODEREG operand of the DFHEIENT macro to establish multiple base registers, must include an LTORG statement to ensure that the literals, generated by either DFHEIENT or a DLI initialization call, fall in the range of the base register.

In general, an LTORG statement is needed for every CSECT that exceeds 4095 bytes in length.

DFHEIRET macro

For AMODE(24) and AMODE(31) programs, the DFHEIRET macro calls the EPILOG program to release the working storage of the application program.

You can specify the following parameter for the DFHEIRET macro:

RCREG
Specify the register into which the return code is placed. If you do not specify a value, a return code is not passed back to the caller of the program. There is no default value.

The translator inserts the DFHEIRET macro, with no parameters specified, immediately before the END statement (unless you specify the NOEPILOG translator option to prevent this). END must be in uppercase for the

translator to recognize it. If the DFHEIRET macro is invoked before this translator-inserted DFHEIRET macro, the translator-inserted macro does not generate any code.

AMODE(64) assembler language programs

CICS Transaction Server supports non-Language Environment assembler language programs that run in 64-bit addressing mode.

SYSSTATE macro

To declare that an application is AMODE(64), you use the z/OS Assembler SYSSTATE (Identify system state) macro.

1. Include a SYSLIB statement for the SYS1.MACLIB macro library (which contains the SYSSTATE macro) in your compile JCL.
2. Ensure one of the following:
   - The SYSSTATE macro with parameter AMODE64=YES is placed immediately after the CICS translator options statement. For example:

     ```
     *ASM   XOPTS(NOPROLOG NOEPILOG)
            SYSSTATE AMODE64=YES
     ```

   - If the program does not specify CICS translator options, the SYSSTATE macro with parameter AMODE64=YES is the first statement.

   The SYSSTATE statement must be on one line, because CICS does not support continuation for this statement. For more information about the SYSSTATE macro, see SYSSTATE — Identify system state in z/OS MVS Programming: Assembler Services Reference, Volume 2.

If the application is declared as AMODE(64) in this way, the following CICS-supplied macros generate AMODE(64) code:

- DFHEIENT
- DFHEISTG
- DFHEIRET
- DFHECALL

If the SYSSTATE macro is not specified, or is specified without parameter AMODE64=YES when any of these CICS-supplied macros first generates code, these macros generate AMODE(24) or AMODE(31) code.

DFHEIENT macro

For AMODE(64) programs, the DFHEIENT macro calls the AMODE(64) PROLOG program, which allocates working storage to hold any user variables and for CICS use. The PROLOG program sets up the CICS portion of this storage. When the PROLOG program returns, this code sets up the registers specified by the DFHEIENT parameters.

You must specify the DFHEIENT macro parameters to specify that your program uses relative addressing instructions, because only relative addressing is supported. For relative addressing, you do not need any base registers to address your program instructions, but you must use at least one register to address static data in your program. Use the STATREG and STATIC parameters to set up one or more static registers.

You can specify the following parameters for the DFHEIENT macro:

CODEREG
Specify a value of 0 (the default) to specify relative addressing.

DATAREG
Specify one or more working storage registers for the application program. The default is register 13, and it is advisable to use register 13 as your first data dynamic-storage register. If you do not, the code generated by the DFHECALL macro adds extra instructions to manipulate register 13. The DFHECALL macro ensures that register 13 addresses the save area that DFHEISTG defined in dynamic storage.

EIBREG
Specify the register to use to address the EXEC interface block (EIB). The default is register 11.

STATREG
Specify one or more static registers for the application program to use. The default is register 3.

STATIC
Specify the assembler label of the start of the static area. You must specify a value; there is no default for this parameter.

Use the NOPROLOG translator option and specify the DFHEIENT macro with the appropriate parameters for relative addressing. If you do not specify the DFHEIENT macro, the translator inserts a DFHEIENT macro without the required parameters and the following error occurs:

```
12,DFHEIENT - AMODE 64 - STATIC REQUIRED
```

The following two example DFHEIENT statements generate the same code. In the first statement, all the parameters are coded (specifying the default values). In the second statement, only the parameter that does not have a default value is coded.

```
DFHEIENT CODEREG=0,DATAREG=13,EIBREG=11,STATREG=3,STATIC=STAT
DFHEIENT STATIC=STAT
```

DFHEIRET macro
For AMODE(64) programs, the DFHEIRET macro calls the AMODE(64) EPILOG program to release the working storage of the application program.

You can specify the following parameter for the DFHEIRET macro:

RCREG
Specify the register into which the return code is placed. If you do not specify a value, a return code is not passed back to the caller of the program. There is no default value.

The translator inserts the DFHEIRET macro, with no parameters specified, immediately before the END statement (unless you specify the NOEPILOG translator option to prevent this). END must be in uppercase for the translator to recognize it. If the DFHEIRET macro is invoked before this translator-inserted DFHEIRET macro, the translator-inserted macro does not generate any code.

For more information about the DFHEISTG storage, see "Extensions to dynamic storage" on page 73. For more information about the DFHECALL and DFHEIRET macros, see .

Related information

EXEC interface design overview

Language Environment coding requirements for assembler language applications

Assembler language programs are classified as either conforming or non-conforming with respect to Language Environment. Conformance depends on the linkage and register conventions observed, rather than the assembler used.

A Language Environment-conforming assembler language routine is defined as one that is 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 language subroutines can be called either statically or dynamically from C, C++, COBOL or PL/I. However, there are differences in register conventions and other requirements for the two types. For example, to communicate properly with Language Environment-conforming assembler language routines, you must observe certain register conventions on entry to the assembler language routine, while it is running, and on exit from the assembler language routine.

Rules for mixing languages, including assembler language, are discussed in "Mixing languages in Language Environment" on page 15.

64-bit addressing mode is not supported for Language Environment-conforming assembler language programs.

For more detailed information, or for explanations of the terms used in this section, see z/OS Language Environment Programming Guide.

Conforming MAIN programs

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

- Use the macros provided by Language Environment. For a list of these macros, see z/OS Language Environment Programming Guide.
- Ensure that the CEEENTRY macro contains the option MAIN=YES. (MAIN=YES is the default).
- Translate your assembler language routine with *ASM XOPTS(LEASM) or, if it contains CICS commands, with *ASM XOPTS(LEASM NOPROLOG NOEPILOG).

Conforming subroutines

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

- Use the macros provided by Language Environment. For a list of these macros, see z/OS Language Environment Programming Guide.
- Ensure that the CEEENTRY macro contains the option MAIN=NO. (MAIN=YES is the default).
- Translate your assembler language 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 Next Available Byte (of

storage). NAB=NO means that this field might not be available, so the CEEENTRY macro generates code to find the available storage.)

Register conventions for entry into conforming routines

On entry into a Language Environment-conforming assembler language subroutine, the following 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 language routine. On entry to an assembler language 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 halfword 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.

Register conventions while a conforming routine is running

R13 must point to the DSA of the routine at all times while the Language Environment-conforming assembler language 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 language routine that is not Language Environment-conforming.
- When calling a Language Environment-conforming assembler language routine that specifies NAB=NO on the CEEENTRY macro.

Register conventions for exit from conforming routines

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

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

Non-conforming assembler language routines running under Language Environment

Observe the following conventions when running non-Language Environment-conforming assembler language routines 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 language 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 *Compiler and Run-Time Migration Guide* for the language in use.

Non-conforming assembler language routines cannot use Language Environment callable services.

Calling assembler language programs

Assembler language application programs can be invoked by COBOL, C, C++, PL/I, or assembler language application programs by using LINK or XCTL commands.

64-bit addressing mode

AMODE (64) assembler language application programs that contain commands can have their own RDO program definition. Such programs can be invoked by COBOL, C, C++, PL/I, or assembler language application programs by using LINK or XCTL commands.

24-bit and 31-bit addressing mode

AMODE(24) and AMODE(31) assembler language application programs that contain commands can have their own RDO program definition. Such programs can be invoked by COBOL, C, C++, PL/I, or assembler language application programs by using LINK or XCTL commands. However, because AMODE(24) and AMODE(31) 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, can 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 more information about mixing languages in an application load module, see “Mixing languages in Language Environment” on page 15. For information about programs with different addressing modes, see “Using mixed addressing modes” on page 306.

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. If you do 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, optionally followed 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 is stored in a library that is included in the SYSLIB data set concatenation.

The load module created by the DFHYITPL procedure includes both the DFHEAI stub (included by DFHEITAL) and the DFHELII stub (included by DFHYITPL). The linkage editor or binder program issues 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 write your own JCL, you must include the DFHELII stub, because this contains the entry points that are 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 passed to the translator, as in the example just given, you do not need to code these macros.

Extensions to dynamic storage

You can extend dynamic storage to provide extra storage for user variables by defining these variables in your source program in a DSECT called DFHEISTG.

24-bit and 31-bit addressing mode

For AMODE(24) and AMODE(31) programs, the maximum amount of dynamic storage obtainable using the DFHEISTG DSECT is 65 264 bytes. DFHEISTG is a reserved name. This storage is initialized to X'00'. At translation, the translator inserts the DFHEISTG macro immediately following your DFHEISTG DSECT instruction. In this way, the DSECT describes dynamic storage needed for the parameter list, for the command-level interface, and for any user variables. At link-edit time, use the STORAGE option of the CEEXOPT macro to ensure that the DFHEISTG storage is initialized to x'00', for example, `CEEXOPT STORAGE=(,,00)`. Make sure that your application propagates or initializes any constants that are defined in the user DFHEISTG area.

The example in Figure 10 on page 74 shows a simple assembler language application program that uses such variables in dynamic storage.

```
Source program

DFHEISTG DSECT
         COPY  DFH$AGA           INPUT MAP DSECT
         COPY  DFH$AGB           OUTPUT MAP DSECT
MESSAGE  DS    CL39
INQUIRY  CSECT
         EXEC CICS RECEIVE MAP('DFH$AGA')
         MVC   NUMBO,KEYI
         MVC   MESSAGE,=CL(L'MESSAGE)'THIS IS A MESSAGE'
         EXEC CICS SEND MAP('DFH$AGB') ERASE
         END


This source program is translated to:

         DFHEIGBL ,                INSERTED BY TRANSLATOR
DFHEISTG DSECT
         DFHEISTG                  INSERTED BY TRANSLATOR
         COPY DFH$AGA              INPUT MAP DSECT
         COPY DFH$AGB              OUTPUT MAP DSECT
MESSAGE  DS    CL39
INQUIRY  CSECT
         DFHEIENT                  INSERTED BY TRANSLATOR
*        EXEC CICS RECEIVE MAP('DFH$AGA')
         DFHECALL =X'1802C000080000000040900000020',
               (CHA7,=CL7'DFH$AGA*'),(______RF,DFH$AGAI)
         MVC   NUMBO,KEYI
         MVC   MESSAGE,=CL(L'MESSAGE)'THIS IS A MESSAGE'
*        EXEC CICS SEND MAP('DFH$AGB') ERASE
         DFHECALL =X'1804C000080000000004E204000020',
               (CHA7,=CL7'DFH$AGB*'),(______RF,DFH$AGBO)
         DFHEIRET                  INSERTED BY TRANSLATOR
         DFHEISTG                  INSERTED BY TRANSLATOR
         DFHEIEND                  INSERTED BY TRANSLATOR
         END
```

Figure 10. Translated code for user variables

64-bit addressing mode

For non-Language Environment AMODE(64) assembler language programs, the DFHEISTG macro generates an AMODE(64) DSECT. DFHEISTG storage is obtained from 31-bit storage (above 16 MB but below 2 GB), not from 64-bit storage. The maximum amount of dynamic storage obtainable using the DFHEISTG DSECT is 65 264 bytes. This storage is initialized to X'00'.

At translation, the translator inserts the DFHEISTG macro immediately following your DFHEISTG DSECT instruction. In this way, the DSECT describes dynamic storage needed for the parameter list, for the command-level interface, and for any user variables.

CICS defines the front portion of DFHEISTG storage.

The EIB pointer, DFHEIBP, and the COMMAREA pointer, DFHEICAP, are set up before the DFHEIENT macro returns control to the application program. These pointers are 64-bit pointers in 24-bit or 31-bit storage.

Developing AMODE(64) assembler language programs

CICS Transaction Server supports non-Language Environment (LE) assembler language programs that run in 64-bit addressing mode (AMODE(64)).

Before you begin

Check the relevant section of "Assembler language programming restrictions and requirements" on page 61.

About this task

The following procedure summarizes the information that relates specifically to developing AMODE(64) assembler language programs.

Procedure

1. Use the z/OS Assembler SYSSTATE macro to declare that an application is AMODE(64). See "Coding the EXEC CICS assembler interface" on page 65.
2. Specify the DFHEIENT macro with the appropriate parameters for relative addressing. See "Coding the EXEC CICS assembler interface" on page 65.

   For information about the DFHEISTG storage, see "Extensions to dynamic storage" on page 73.
3. If you are not using EXEC CICS RETURN, specify the DFHEIRET macro, or use the translator-inserted version. See "Coding the EXEC CICS assembler interface" on page 65.
4. Translate, assemble and link-edit the program. See "Translating, assembling, and link-editing assembler language application programs" on page 135.

   For information about the DFHECALL macro, see .

   For information about link-editing the stub programs, see "Using the EXEC interface modules for AMODE(64) applications" on page 104.

Related information

z/OS MVS Programming: Assembler Services Guide

Part 3. Translating, compiling, installing and testing application programs

How to translate, compile and install application programs in CICS, and what facilities are provided for testing applications.

© Copyright IBM Corp. 1989, 2013

Chapter 7. Translation and compilation

Some older compilers (and assemblers) cannot process CICS commands directly. 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.

Most compilers 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, many of the translation tasks are done at compile time and you do not need to complete the additional translation step. For more information about the task in the translation step, see "The translation process" on page 81.

This section describes:

- "The integrated CICS translator"
- "The translation process" on page 81
- "The CICS-supplied translators" on page 84
- "Using a CICS translator" on page 94
- "Defining translator options" on page 95
- "Using COPY statements" on page 98
- "The CICS-supplied interface modules" on page 98
- "Using the EXEC interface modules for AMODE(24) and AMODE(31) applications" on page 99

The integrated CICS translator

Using the integrated translator, you can translate and compile your high-level source code in a single step. Compilers that support the integrated translator scan the application source and call the integrated translator at relevant points. The integrated translator converts **EXEC CICS** commands into comments and generates CALL statements appropriate to the language.

There are versions of the integrated translator for the following languages:

C
C++
COBOL
PL/I

With the integrated translator, application development is faster because there is no separate translation step. It is also made easier because there is only one listing; the original source statements and the CICS error messages are included in the compiler listing. The CICS-supplied separate translators change the line numbers in source programs, which means that with translator-generated calls you need an intermediate listing that must be used when debugging an application program.

With the integrated translator, the process of translating and compiling is also less error-prone because it is no longer necessary to translate included members separately.

© Copyright IBM Corp. 1989, 2013

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

The releases of the language compilers that support the integrated translator are listed in High-level language support. If you use any other compiler, including Assembler, you must translate your program before compiling it.

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 COBOL for z/OS, C/C++ for z/OS, and for Enterprise PL/I for z/OS.

About this task

The procedure that you use needs to have CICSTS51.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 remain in the user region for integrated CICS translator services. Instead, use a value such as SIZE(4000K), which should work for most programs.

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

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 z/OS Application Programming and SQL Guide* for more information on using the SQL statement coprocessor.

Specifying CICS translator options

You can specify CICS translator options when you use the PL/I, COBOL, XL C, or C++ compiler.

About this task

For a description of all the translator options, see “Defining translator options” on page 95.

Many translator options do not apply when using the integrated CICS translator, for example those associated with translator listings. If these options are specified, they are ignored. The EXCI option is not supported; the CICS option is assumed.

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

- 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

Procedure

- 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 parentheses. For example:

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

  For more information about specifying PL/I compiler options, see *Enterprise PL/I for z/OS 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 parentheses. 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 about specifying COBOL compiler options, see *Enterprise COBOL for z/OS: Programming Guide*.
- To specify CICS translator options when using the XL C and C++ compiler specify the compiler option, CICS, with the translator options inside parentheses and separated by commas. For example:

  ```
  CICS(opt1,opt2,optn ...)
  ```

  Alternatively, you can specify translator options on a `#pragma` statement in the program source on the XOPTS or CICS keyword.

  For more information about specifying C and C++ compiler options, see *z/OS XL C/C++ User's Guide*.

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 EXEC CICS, EXEC CPSM and EXEC DLI statements. For compilers with integrated translators, the compilers call the CICS translator directly to handle the translation of these commands.

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 11 shows these 3 steps.

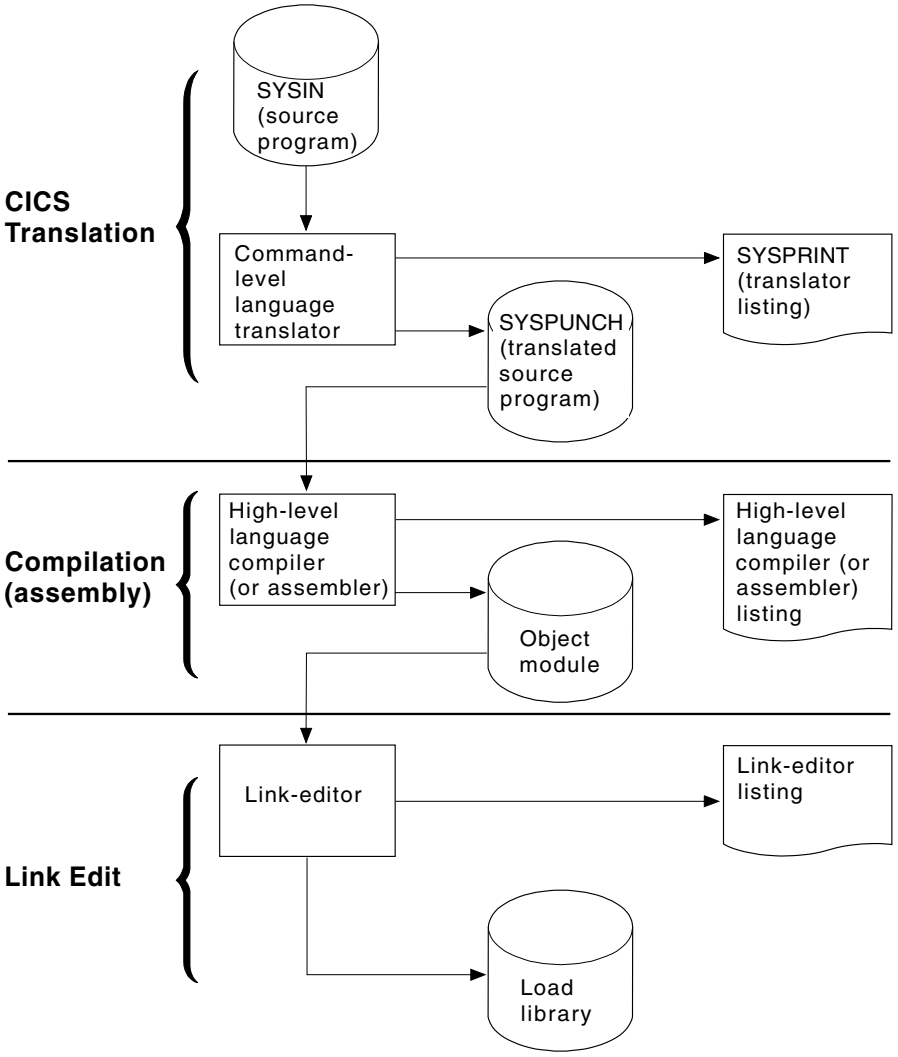


Figure 11. Preparing an application program

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 133 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” 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 98.

Related concepts:

“Using a CICS translator” on page 94
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.

Related reference:

“The CICS-supplied translators”
The CICS-supplied translators are installed in the CICSTS51.CICS.SDFHLOAD library.

The CICS-supplied translators

The CICS-supplied translators are installed in the CICSTS51.CICS.SDFHLOAD library.

The following table lists the translators in this library.

Table 5. CICS-supplied translator by language

Language	Translator
Assembler	DFHEAP1$
C	DFHEDP1$
COBOL	DFHECP1$
PL/I	DFHEPP1$

Related concepts:

“The translation process” on page 81
For compilers without integrated translators, CICS provides a translator program for each of the languages in which you may write, to handle EXEC CICS, EXEC CPSM and EXEC DLI statements. For compilers with integrated translators, the compilers call the CICS translator directly to handle the translation of these commands.

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.

About this task

If you use ATTACH, LINK, or XCTL, use the appropriate translator load module, DFHExP1$, 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 options

The command-level language translator has a list of options that you can select from to translate your COBOL, C, C++, PL/I, and Assembler programs. Each translator option is explained and lists the languages that you can use it with.

APOST

(COBOL only)

APOST indicates that literals are delineated by the apostrophe or a quotation mark. QUOTE is the alternative option, which indicates double quotation marks. The same value must be specified for the translator step and the following compile step.

The supplied COBOL copybooks are generated using this option and have a single quotation mark. If you are using any supplied copybooks in your application to interface to a CICS component, ensure that you use the APOST option, not the QUOTE option.

CBLCARD

(COBOL only) Abbreviation: CBL

CBLCARD specifies that the translator is to generate a CBL statement. CBLCARD 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 you specify both COBOL2 and COBOL3, 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 options are mutually exclusive. For information about which Language Environment-conforming compilers are available, see Chapter 2, "Programming languages and Language Environment," on page 11.

CPP
(C++ only)

CPP specifies that the translator is to translate C++ programs for compilation by a supported C++ compiler.

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 might contain double-byte characters. The DBCS option 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, use RETURN and specify NOEPILOG.

EPILOG is the default; the alternative NOEPILOG prevents the translator from inserting the macro DFHEIRET. (See the *CICS Application Programming Reference* 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). EXCI API 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.

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). The alternative is NOFEPI. For more information about FEPI, see FEPI application development in Developing applications.

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 that 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 CICS API commands in Reference -> Application development.

GRAPHIC

(PL/I only)

GRAPHIC specifies that the source program might contain double-byte characters. The GRAPHIC option 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. Prevent unintended shift-out and shift-in codes 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 language 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 subroutines in a CICS environment. The LEASM option 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 are in 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 with LEASM" on page 102.

LENGTH

(COBOL, assembler language 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 inserts a USING DFHEIBLK DFHCOMMAREA statement in the PROCEDURE DIVISION, if one does not exist, and ensures that the LINKAGE SECTION (creating one if necessary) contains definitions for DFHEIBLK and DFHCOMMAREA.

LINKAGE is the default; the alternative is NOLINKAGE.

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 MARGINS 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 as follows:

m Column number of left margin.

n Column number of right 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 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 one line before printing.

0 Skip two lines before printing.

- Skip three 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 which language is to be used for the translator message output:

CS Simplified Chinese.

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. Ensure that RENT, NODYNAM, and LIB are specified.

NOCPSM
NOCPSM specifies that the translator is not to process EXEC CPSM commands. NOCPSM 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, use **EXEC CICS RETURN** and specify NOEPILOG. NOEPILOG prevents the translator inserting the macro DFHEIRET. The alternative is EPILOG, which is the default.

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 language 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.

The NOLINKAGE option 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.

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 unrecoverable errors; instead, a dump is produced. 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.) NOVBREFis 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 margins 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 might be necessary to change the input margins for the compiler being used. If the OPMARGINS value is allowed to default, it is not necessary to change the input margins for the compiler being used.

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.

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 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 that 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 are rejected at translate time. These commands are ACQUIRE, COLLECT, CREATE, DISABLE, DISCARD, ENABLE, EXTRACT, INQUIRE, PERFORM, RESYNC, and SET. Theses commands are used by system programmers. For programming information about these commands, see System commands in Reference > System programming.

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 unrecoverable 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. Use this option only in special situations because it imposes restrictions on the programs using it. A program translated with the SYSEIB option must:

- Execute in AMODE(31), because 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).
- The use of the SYSEIB option implies the use of the NOHANDLE option on all CICS commands issued by the program. Commands can 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.)

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 are later 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. See "Defining translator options" on page 95.

You can specify your choices in the following ways:

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

Table 6. Statement provided for each language

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 in the procedure is TRN, the following example statement sets translator options for a COBOL program:

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

If you use one method to specify an option and use the other method to specify the same option, or an option that conflicts, 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 can 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))
```

The following 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 can 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)
```

The following form 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 language 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.

Related concepts:

“The translation process” on page 81
For compilers without integrated translators, CICS provides a translator program for each of the languages in which you may write, to handle EXEC CICS, EXEC CPSM and EXEC DLI statements. For compilers with integrated translators, the compilers call the CICS translator directly to handle the translation of these commands.

Defining translator options

You can specify the translator options that apply to all languages except where stated otherwise.

Table 7 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 84). If you want an option that is not the default, you must specify it, as described in "Using a CICS translator" on page 94.

Translator options table

The translator options are shown, in tabular form, along with the program languages to which they apply.

Table 7. 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 **NOCPSM**	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

Table 7. Translator options applicable to programming language (continued)

Translator option	COBOL	C	C++	PL/I	Assembler
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.

About this task

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 CICS facilities.

Application programs require a stub 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 CICSTS51.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 CICSTS51.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 CICSTS51.CICS.SDFHLOAD library, must be link-edited with each application program that uses the SAA Resource Recovery facility.

Using the EXEC interface modules for AMODE(24) and AMODE(31) applications

For AMODE(24) and AMODE(31) applications, the CALL statements that the language translators generate invoke EXEC interface modules that provide communication between your code and the CICS EXEC interface program, DFHEIP.

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*; that is, function-dependent sections of code used by the CICS high-level programming interface. The stub, which is provided in the SDFHLOAD library, must be link-edited with your application program. These stubs are invoked during execution of EXEC CICS and EXEC DLI commands. Stubs are provided for each programming language.

Table 8. Interface modules (stubs)

Language	Interface module name
Assembler	DFHELII and DFHEAI0
All HLL languages and Assembler MAIN programs that use 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 compatible with earlier and later versions, and CICS application modules never need to be relinked to include a later level of any of these stubs.

Except for DFHEAI0, these stubs all provide the same function, which is to provide a linkage from EXEC CICS commands to the required CICS service. To do this, the stubs provide various entry points that are called from the translated EXEC CICS commands, and then execute 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 old versions of the CICS PL/I translator. It contains the entries DFHEXEC (for C and C++ application programs), DFHEI1 (for COBOL and Assembler), and DFHEI01 (for PL/I).

Each stub begins with an 8 byte eyecatcher in the form DFHY*xnnn*, where *x* indicates the language supported by the stub (A indicates 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 DFHE*xxxx*, 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 5 Release 1 is DFHYI680.

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

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 it processes 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('xxxxxxxxxxxxxxxxx' /* '0E 08 80 00 03
00 00 10 00 F0 F0 F0 F0 F0 F0 F1 F0 'X */, NEXT);
END;
```

In the previous example, 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++ program, 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'0E0800000800001000'
```

The assembly of the DFHECALL macro invocation as shown 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'0E08000008001000'
         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 5 Release 1 is DFHYA680, with the release numbers indicating this stub was supplied with CICS Transaction Server for z/OS, Version 5 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.

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.

Example assembler language program with LEASM

An example CICS assembler program and the corresponding code after it is translated demonstrate the result of translating an assembler program using the LEASM option. This translator option generates code for a Language Environment-conforming assembler MAIN program.

Figure 12 shows a simple CICS assembler language 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 12. A simple CICS assembler language program.

Figure 13 on page 103 shows the code after the program is translated and assembled:

```
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                   Reserved                             02-CEEDS
CEEDSAAUTO DS  0D                  Automatic storage starts here        02-CEEDS
CEEDSAEND  DS  0D                  End of DSA                           02-CEEDS
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
```

Using the EXEC interface modules for AMODE(64) applications

For non-Language Environment AMODE(64) assembler language programs, the CALL statements that the language translator generates invoke EXEC interface modules that provide communication between your code and the CICS EXEC interface program, DFHEIG.

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*; that is, function-dependent sections of code used by the CICS high-level programming interface. These stubs, which are provided in the SDFHLOAD library, must be link-edited with your application program. These stubs are invoked during execution of EXEC CICS commands.

The following stub programs are provided to transfer control from an AMODE(64) application program to CICS. These stubs use CICS internal control blocks to locate the required CICS code.

- DFHEAG0 (prolog and epilog stub).

  When the DFHEIENT macro calls this stub, it transfers control to the AMODE(64) PROLOG program.

  When the DFHEIRET macro calls this stub, it transfers control to the AMODE(64) EPILOG program.
- DFHEAG (command stub).

  When the DFHECALL macro calls this stub, it transfers control to the AMODE(64) initial command processor. DFHEG1 is an alias of DFHEAG and the DFHECALL macro actually calls the alias, DFHEG1.

You must link-edit the DFHEAG and DFHEAG0 stubs with the AMODE(64) application program. Use the following binder statements:

```
ORDER  DFHEAG
INCLUDE  SYSLIB(DFHEAG)
ENTRY  program_name
NAME  program_name(R)
```

program_name is the name of the AMODE(64) application.

This is similar to the way that stubs DFHEAI and DFHEAI0 are link-edited with AMODE(24) and AMODE(31) assembler language application programs.

Related topics

DFHEAG

DFHEAG0

Chapter 8. Deploying applications to a platform

You can set up a platform to deploy and manage CICS applications over multiple CICS regions. Use the CICS Explorer® or the CICS Explorer SDK to create a platform, applications, and policies to manage the applications.

About this task

The steps in this task summarize the workflow described in the section "Working with platforms, applications, and policies" in the *CICS Explorer User Guide*, in the CICS Explorer help. If you are developing new applications to deploy to platforms, follow this workflow. The sections "Working with platforms, applications, and policies" and "Working with bundles" in the *CICS Explorer User Guide*, in the CICS Explorer help, have more detailed instructions for each step in this task.

If you want to start moving existing applications towards cloud enablement, you might prefer to follow the progressive adoption roadmap described in Getting started with cloud enablement.

Procedure

1. Create and install a platform. See Setting up a platform.
2. For each application that you want to deploy to the platform, create CICS bundles that define or import the components of the application. Do not deploy the CICS bundles yet, because you deploy them as part of the application. See Defining CICS bundles.
3. Create policies containing policy rules to manage the applications in your platform. Policy rules describe controls or actions that can be applied to one or more application tasks that run in a platform. A policy is defined and deployed in a CICS bundle. See Policies.
4. For each application that you want to deploy to the platform, create and export an application bundle and an application binding. See Packaging applications for deployment.
5. Install and enable each application that you want to deploy to the platform. See Installing and enabling an application.
6. Use the application context data generated for your applications to monitor and measure how much resource each application is using. You can add further policy rules to automatically respond when resource usage exceeds thresholds. See Measuring the resource usage of applications.
7. As your capacity requirements increase or decrease, add CICS regions to the platform or remove them from it. See Administering platforms and applications.
8. Update applications in the platform as required, and disable or discard them if they are no longer required. See Administering platforms and applications.

Installing an application in a platform

To deploy a packaged CICS application to a platform, you define and install an application definition (APPLDEF) resource. The APPLDEF resource defines the application in the data repository, and provides lifecycle management capabilities.

© Copyright IBM Corp. 1989, 2013

Before you begin

A platform must be configured in the CICSplex. The application must be packaged as an application bundle, as described in Packaging applications for deployment in Developing applications, and an application binding must exist for the application.

About this task

Using the CICS Cloud perspective in the CICS Explorer, you can define and install CICS applications in a platform. The application provides a single point of management for enabling, disabling, and updating the application resources across the CICS regions in the platform. See the section "Working with platforms, applications, and policies" in the *CICS Explorer User Guide* for detailed help on how to perform each step.

Procedure

1. Ensure that the application bundle and application binding are deployed on the correct platform home directory in the zFS file system. The CMAS must have read and execute access to this directory.
2. Create an APPLDEF in the CICSPlex SM data repository that points to the application bundle and application binding in zFS. The application definition identifies the target platform where you want to run the application. You can edit the application definition after you have created it and before it is installed, but you cannot edit it after it is installed.
3. Install the application definition. CICSPlex SM creates an APPLCTN resource to represent the application in the CICSplex, and also creates a record for the application in the data repository, which is used in recovery processing for the bundles for applications. CICSPlex SM uses the information in the application bundle and the application binding to install the CICS bundles in the CICS regions in the platform. The application is initially installed in a disabled state.
4. Enable the application. CICSPlex SM enables the CICS bundles that were installed for the application, so that the application is available for use in all the CICS regions.

Results

When you install an application in an active platform, CICSPlex SM installs the CICS bundles immediately in all the CICS regions that are defined as part of the platform, and available at the time when you install the application definition. Each BUNDLE resource is dynamically created and is given a unique name. The resources that are defined inside each CICS bundle are dynamically created in the CICS regions during the BUNDLE resource installation. CICSPlex SM also installs the CICS bundles in CICS regions in the platform if you start or restart the regions after the time when you install the application definition. If you have enabled the application, CICSPlex SM installs the bundles in started or restarted regions in an enabled state.

The relationship between the application and each installed CICS bundle is saved in a management part. The management part is a MGMTPART record that is created automatically for each CICS bundle during the application install process. It records the CICS regions where the bundle is installed, and tracks the status of the bundle in the CICS regions.

What to do next

Click **Operations** > **Cloud Explorer** to view the installed application and check its overall status. In the application hierarchy, expand the region types and bundles for the application to check on the status of each CICS bundle. The application is enabled if the CICS bundles referenced by the management parts for the application have been installed in the CICS regions, and all of them are enabled. See Checking the status of an application for instructions to check the status of an application, and for what to do if the application is not enabled.

Getting started with cloud enablement

If you want to move toward cloud enablement for your CICS applications, you can use the phased approach described here to set up a platform, deploy the application to the platform, manage and monitor the application by using a policy and application context data, and remove the application at the end of its lifecycle.

For agile service delivery, CICS resources are packaged together, hosted as applications on a platform, and managed dynamically by using policies. Delivered in this way, the resources can be moved through their lifecycle with greater ease and flexibility, so that you can respond to changes in business and user requirements with increased speed and confidence. Agile service delivery moves you towards a service delivery platform for cloud computing.

The workflow described in this process is an adoption roadmap to progressively implement platforms, applications, and policies to move existing applications towards cloud enablement. If you are developing new applications to deploy on platforms, follow the workflow described in the section "Working with platforms, applications, and policies" in the *CICS Explorer User Guide*, in the CICS Explorer help. That procedure describes how to use the CICS Cloud perspective to set up a platform and create suitable applications and policies.

If you want to try out the complete process first by using one of the sample applications supplied with CICS, you can follow through Scenario: Deploying an application to a platform.

Stage 1: Create a platform

You use a platform to deploy and manage CICS applications over multiple CICS regions. The platform provides services to the applications. In this stage of cloud enablement, you can create a platform. You can create new region types in your platform to contain new CICS regions where your CICS applications will be installed. Your platform can also adopt CICS regions where your existing CICS applications are installed. To create a platform, you must be using CICSPlex SM to manage your CICS regions; see Getting started with CICSPlex SM.

Create a platform shows the process for creating a platform and deploying it in a CICSplex:

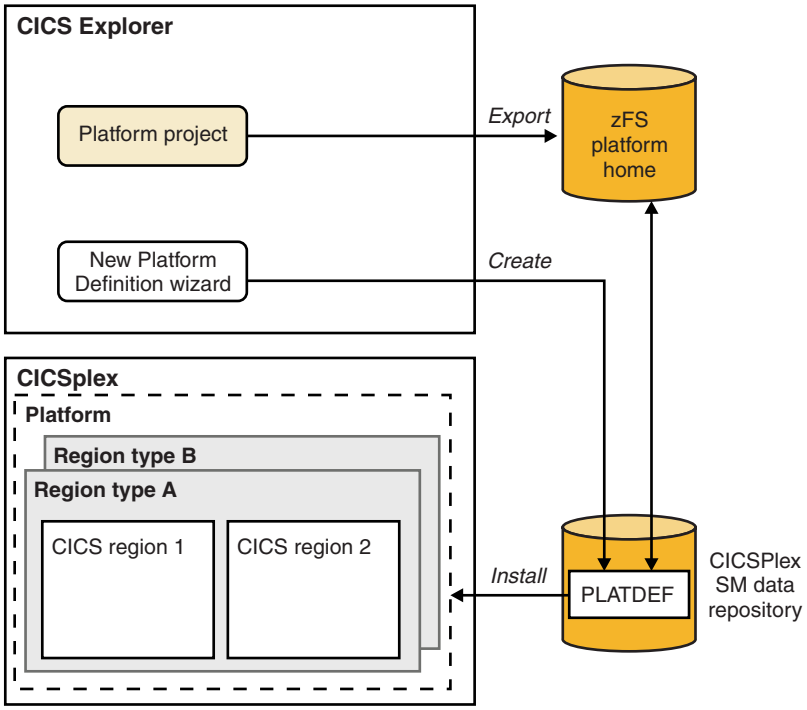


Figure 14. Stage 1: Create a platform

In the CICS Explorer, or the CICS Explorer SDK, use the Project Explorer view in the CICS Cloud perspective to create a CICS Platform project and export it to the z/OS UNIX file system. The CICS Platform project defines a platform bundle, which is a type of management bundle that describes the platform, including the CICS bundles that are deployed with it and the region types that are part of it. A platform can include a mix of created region types, where you define a region type and set up new CICS region definitions, and adopted region types, where you adopt an existing CICS system group (CSYSGRP) as a region type with its existing CICS regions. Use the Platform editor in the CICS Explorer to work with your platform bundle and confirm the region types before you export the project.

After you have exported the CICS Platform project, use the CICS Explorer to create and install a platform definition, which is a PLATDEF resource definition. When you install the platform definition, CICS creates the platform in the CICSplex. You can add or remove individual CICS regions in the region types in an installed, active platform.

Setting up a platform explains how to design and create a platform. The section "Working with platforms, applications, and policies" in the *CICS Explorer User Guide*, in the CICS Explorer help, has instructions to create a CICS Platform project and a platform definition.

Stage 2: Create an application

When you package a CICS application in an application bundle, you can deploy, manage, and monitor the application as a single entity. With an application bundle, you can install and manage the resources for the application in all of the CICS regions in a platform. In this stage of cloud enablement, you can package your existing CICS application into an application bundle for deployment on a platform.

Figure 15 shows the process for creating an application:

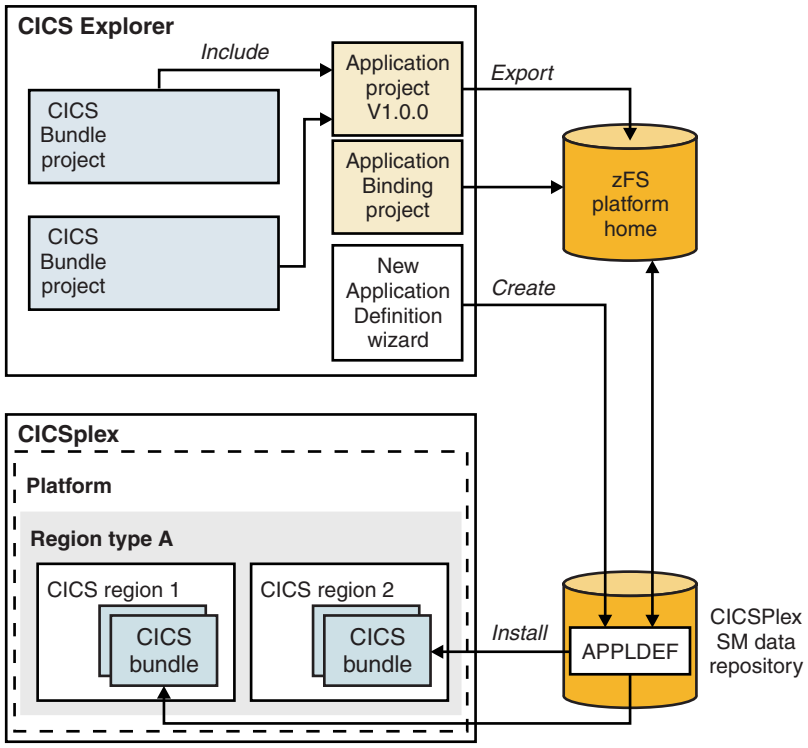


Figure 15. Stage 2: Create an application

Use the CICS Explorer to create one or more CICS bundles to contain the resources for each component of the application. The CICS bundles also declare any dependencies on other resources outside the bundles, known as imports, which must be available in the CICS regions for the application to run. At this stage, you can specify all the installed resources for your existing CICS application as imports. At a later stage, you can modify the bundles so that the resources are created when the bundles are installed.

The section "Working with bundles" in the *CICS Explorer User Guide*, in the CICS Explorer help, has instructions for creating CICS bundles for the components of the application. CICS bundle projects must have an ID and a version number to be included in an application project. If you want to use any CICS bundle projects from releases before CICS Transaction Server Version 5 Release 1, that do not have an ID and a version number, update the bundle manifest files for these projects to add that information.

When you have created suitable CICS bundles, use the Project Explorer view in the CICS Cloud perspective in the CICS Explorer or the CICS Explorer SDK to create a CICS Application project. The CICS Application project defines an application bundle that references all the CICS bundles for the application.

Map the CICS bundles for the application to region types in the target platform by creating an application binding, in a CICS Application Binding project. Depending on the architecture of your application, you can deploy additional CICS bundles by using the application binding, to customize the behavior of the application for the target platform. As another alternative, CICS bundles can be added to your installed platform.

Export the application bundle and the application binding to the zFS home directory of the platform where you want to run the application. Next, use the Cloud Explorer view in the CICS Cloud perspective in the CICS Explorer to create and install an application definition. The APPLDEF resource definition points to the application bundle and the application binding in the home directory for the platform in zFS. When you install the application definition, CICS installs the CICS bundles into the CICS regions in the platform. You can then enable the application so that it becomes available for use in the CICS regions.

Packaging applications for deployment explains how to package an application in an application bundle. The section "Working with platforms, applications, and policies" in the *CICS Explorer User Guide*, in the CICS Explorer help, has instructions to create an application bundle, application binding, and application definition.

Stage 3: Add application entry points

Application context data is part of the task association data, and it is output as part of the monitoring data for a task. Application context data identifies tasks that are started in the context of your application and platform. You can use the information to measure resource consumption by applications (or by particular routes into applications), to use policy-based management for applications, and to relate tasks to specific applications to help with problem diagnosis.

In this stage of cloud enablement, you can define an application entry point for each of the ways in which your application can be started, that is, for the relevant PROGRAM resources. When a task that does not have an application context calls a program that has an application entry point, CICS creates an application context that becomes associated with the task, and with any subsequent programs that it calls and tasks that it starts.

Figure 16 on page 111 shows the process for adding application entry points:

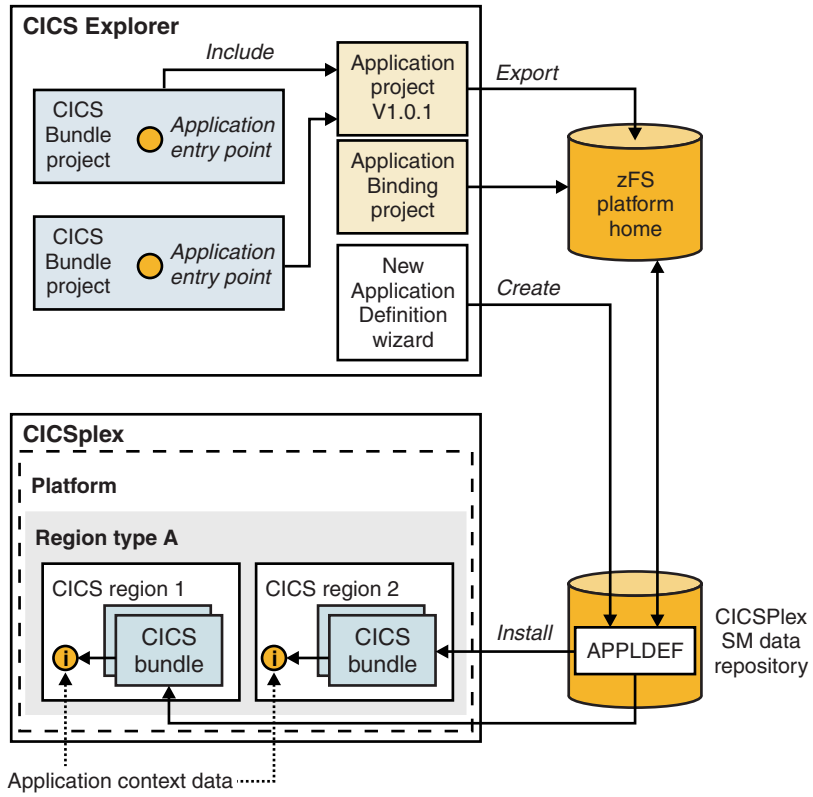


Figure 16. Stage 3: Add application entry points

Use the CICS Explorer or the CICS Explorer SDK to edit one or more of your CICS bundles to add application entry points. You can create application entry points for CICS programs that are part of the application. The application entry point specifies the operation that is taking place in the application, such as a read, write, or update operation.

To see the application context data for your tasks, you need to update and reinstall the application. Application bundles include version information for the application, which you can use to manage applications during testing and promotion from your test environment to your production environment. Use the Project Explorer and Cloud Explorer views in the CICS Cloud perspective to update the application with the new versions of your CICS bundles, apply a new version number to the application and the application binding, and upload the new versions of the application and application binding to the platform home directory in zFS.

When you install the new version of the application into the platform, the CICS regions generate application context data for the tasks relating to your application. The application context data identifies the platform, the application, the application version, and the operation.

The section "Working with platforms, applications, and policies" in the *CICS Explorer User Guide*, in the CICS Explorer help, has instructions for defining application entry points, and for updating applications.

Stage 4: Add resources for the application

When you completed the previous stage of cloud enablement, the CICS regions where your existing CICS application was installed were defined as part of a platform, and your existing CICS application was defined as an application in the platform. However, the resources for the application were defined and managed in the CICS regions in the same way as before. In this stage, you can transfer the responsibility for creating, installing, and managing the resources to the platform.

You can use CICS bundles to package suitable resources for artifacts such as a CICS program or transaction, a CICS library, an Atom feed, an XML-based service, a Java application that has been packaged as an OSGi bundle, or an application or system event. You can only add user-defined resources to a CICS bundle; you cannot add CICS-supplied resources (beginning with DFH). Defining CICS bundles has information about the types of resources that you can define in a CICS bundle, and how they are managed when you have defined them.

Defining resources in a CICS bundle is an important step, because the management and lifecycle of those resources is delegated to the CICS bundles and management bundles that installed the resources. You no longer modify the resources or change their state individually, because they are automatically added, updated, or removed when you operate on the CICS bundles and the application. Application architects must therefore carefully consider which set of resources for an application should have their lifecycle tied to the lifecycle of a CICS bundle. Review the information in Characteristics of bundled resources to help you choose which resources to define in CICS bundles and how to arrange them. If your application uses resources that cannot be defined in a CICS bundle, or that you do not want to define in a CICS bundle, you can continue to specify these as imports.

Figure 17 on page 113 shows the process for adding resources for the application:

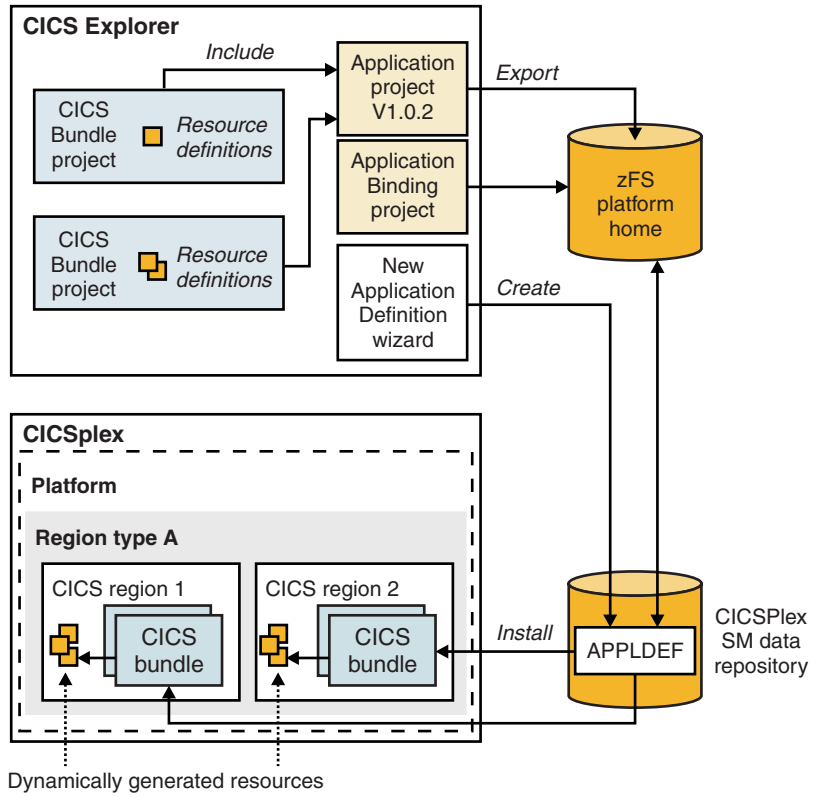


Figure 17. Stage 4: Add resources for the application

Update the CICS bundles for your application to add suitable resources to be dynamically created in the CICS regions when you install the application in the platform. For example, you can define the TRANSACTION and PROGRAM resources for the application in the CICS bundles.

Update the application with the new versions of your CICS bundles, apply a new version number to the application and the application binding, and install the new version of the application in the platform. The resources that you defined in the CICS bundles are dynamically created in the CICS regions. These resources must now be managed through the CICS bundles. CICS automatically manages the deployment of applications when you start or restart CICS regions that are in the platform.

The section "Working with bundles" in the *CICS Explorer User Guide*, in the CICS Explorer help, has instructions to add resources to a bundle.

Stage 5: Add a policy

A policy is an XML definition that contains policy rules to describe controls or actions which can be applied to one or more application tasks. The policy rules define threshold conditions to manage the behavior of user tasks. For example, you can create a rule to define a threshold for the amount of storage allocated by a user task, and make CICS issue a message if the threshold is exceeded. In this stage of cloud enablement, you can create and deploy a policy to manage the tasks relating to your application.

Figure 18 shows the process for adding a policy:

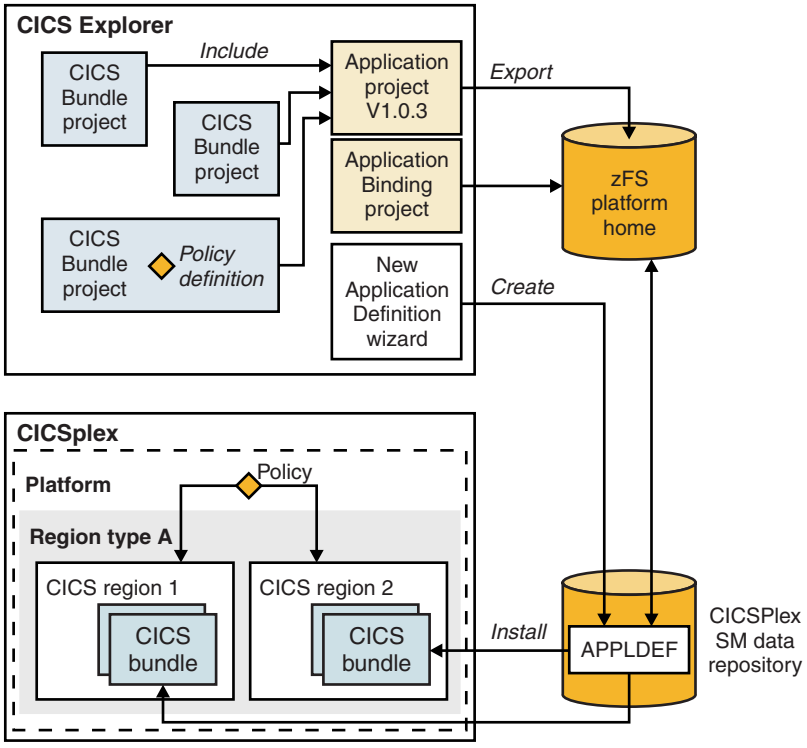


Figure 18. Stage 5: Add a policy

In a CICS bundle, define a policy with suitable policy rules for your existing CICS application. You can add the CICS bundle containing the policy to your application bundle, or to target the policy to a specific environment such as your production CICS regions, add the policy to your application binding instead. If you want to apply the policy to all applications in the platform, add it directly to the platform.

If you deployed the policy with your application bundle or application binding, update and apply a new version number to the application and the application binding, and install the new version of the application in the platform. The policy now applies to the user tasks that have application context for your application, that is, the tasks for which you created application entry points.

The section "Working with platforms, applications, and policies" in the *CICS Explorer User Guide*, in the CICS Explorer help, has instructions to create a policy.

Stage 6: Remove the application from the platform

When you no longer require your existing CICS application, you can complete its lifecycle by removing it from the platform. When you disable and discard an application, the CICS bundles that are deployed as part of the application are also disabled and discarded in the CICS region types to which they are bound.

In the CICS Explorer, or the CICS Explorer SDK, use the Cloud Explorer view in the CICS Cloud perspective to first disable, then discard the application. If you do not require the application again, you can remove the application definition from the data repository. You can also use the z/OS perspective in the CICS Explorer to remove the application bundle from the platform home directory in zFS.

Administering platforms in Administering has instructions to disable, discard, and remove an application from a platform.

Chapter 9. Installing application programs

Applications can either be installed independently, or though the bundles mechanism, a process which dynamically creates the required CICS resources.

Independent applications

To install an application program to run under CICS you must translate and compile your source statements and link-edit 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).

Your application programs can include the following elements:

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

If you are developing application programs to use the CICS dynamic transaction routing facility, use the CICS Interdependency Analyzer tool to detect whether the programs are likely to cause inter-transaction affinity. For more information see "Affinity" on page 312.

Applications as bundles

A bundle is a directory that contains artifacts, and a manifest that describes the bundle and its dependencies. The bundle is the unit of deployment for an application, and specifies a set of resources that CICS creates dynamically when the bundle is deployed. The bundle can also specify any prerequisite system resources required by the application. You can create CICS bundles either using the IBM CICS Explorer or Rational® Developer for System z®.

The separation of resources and prerequisites means that you can install an application into multiple CICS regions without having to repackage or redeploy the bundle.

This section includes:

- "Program installation steps" on page 118
- "Defining MVS residence and addressing modes" on page 126
- "Running application programs in the read-only DSAs" on page 129
- "Using BMS map sets in application programs" on page 132
- "Using the CICS-supplied procedures to install application programs" on page 133
- "Including the CICS-supplied interface modules" on page 135
- "Translating, assembling, and link-editing assembler language application programs" on page 135
- "Installing COBOL application programs" on page 137
- "Installing PL/I application programs" on page 141

© Copyright IBM Corp. 1989, 2013

- "Installing C application programs" on page 141
- "Using your own job streams" on page 145

Program installation steps

There are a number of steps that need to be performed to install an application program to run under CICS.

About this task

The steps are as follows:

Procedure

1. If your compiler does not translate CICS commands, you must translate the program source code to turn CICS commands into calls that are understood by the compiler.
   a. If your program does not use CICS commands and is invoked only by a running transaction (and never directly by CICS task initiation), you do not need a translator step.
   b. You must also translate CICS command-level programs that access DL/I services through either the DL/I CALL or EXEC DLI interfaces. Applications that access DB2 services using the EXEC SQL interface need an additional precompilation step.
2. Compile your program source to produce object code.
3. Link-edit the object module to produce a load module, which you store in an application load library in the DFHRPL or dynamic LIBRARY concatenation. You need additional INCLUDE statements for applications that access DB2 services using the EXEC SQL interface.
4. Create resource definition entries, in the CSD for any transaction that calls the program, and install them.
5. Create a resource definition entry in the CSD for the program using one of the following methods:
   - Using program autoinstall.
   - Using RDO.

Using dynamic program LIBRARY resources

For an application to run, the load module has to reside in a data set in a CICS load LIBRARY concatenation.

CICS has two types of load LIBRARY concatenations:

- The static load LIBRARY concatenation: DFHRPL.
- One or more dynamically defined load LIBRARY concatenation.

Static LIBRARY concatenation- DFHRPL

The startup JCL defines the static load LIBRARY concatenation, DFHRPL, to CICS. DFHRPL contains critical data sets, which must be available for CICS to start up and run, and application program entities. When CICS is running, changes to the DFHRPL data set names are not possible without stopping and restarting CICS. Such changes are not usually an option in today's continuous availability environment.

DFHRPL data set names must conform to the MVS data set naming convention.

Dynamic program LIBRARY concatenation

You can define program LIBRARY concatenations to CICS dynamically. The use of dynamic LIBRARY concatenations provides a number of advantages for the system programmer and the organization:

- They contain one or more data sets from which program artifacts can be loaded.
- You can bring new applications for deployment into service at any time without affecting continuous availability.
- You can easily withdraw existing applications in dynamic LIBRARY concatenations from service without affecting continuous availability.
- You can install patches to existing applications by installing them in a LIBRARY concatenation with a higher ranking than the existing LIBRARY, without affecting continuous availability.
- You can take offline data sets in dynamic LIBRARY concatenations for compression without affecting continuous availability.

LIBRARY data set names must conform to the MVS data set naming convention and you can use alias data sets.

You do not have to use dynamic program LIBRARY concatenations. You may use DFHRPL. In fact, you must define the following data sets in DFHRPL

- SDFHLOAD
- Phase 1 PLT programs
- Non-SMS managed data sets
- Data sets with DISP other than SHR

You can install or create dynamic LIBRARY concatenations as either enabled or disabled.

Enabled

When you install or create a LIBRARY with an enabled status of enabled, CICS attempts to allocate and then concatenate the data sets, before finally opening the LIBRARY concatenation. If any of these steps fails, those that had already succeeded are undone and the LIBRARY is installed as disabled. A message indicates the step that has failed.

Disabled

When you install or create a LIBRARY with an enabled status of disabled, CICS does not attempt to allocate or concatenate the data sets. When the data sets are available and the LIBRARY is ready for use, perform a SET LIBRARY ENABLED command to allocate and concatenate the data sets and open the LIBRARY.

If any of the enable steps in the SET LIBRARY ENABLED operation fails, those that had already succeeded are undone and the LIBRARY remains disabled. A message indicates the step that has failed.

The following examples help you to use dynamic program LIBRARY concatenations.

Examples of using dynamic LIBRARY resources

You choose whether to adopt dynamic LIBRARY resources. You might use them in a test environment, or a production environment, or both. You may decide to move some data sets out of DFHRPL and define them dynamically and to use a combination of DFHRPL and dynamic program LIBRARY resources.

Good candidates for defining in a dynamic LIBRARY are vendor packages or in-house applications that are supplied in one or more data sets.

The following examples show how you can use dynamic program LIBRARY concatenations to manage your programs.

Applying an emergency fix to a CICS system

Install a temporary LIBRARY containing program fixes, to a CICS region.

About this task

- A version of an application being used by the CICS system has a problem that needs correcting.
- An updated version of the application has been created.
- You cannot restart CICS at this time to apply the fix.
- The person who performs this action has the appropriate access authority.

Procedure

1. Add programs and other artifacts which provide the fix to a PDS or PDSE data set, or set of data sets.
2. Use the Business Application Services (BAS) component of CICSPlex SM, or use CEDA or DFHCSDUP or EXEC CICS CREATE, to define a LIBRARY resource that includes the data set, or data sets, containing the fixes. It must have a ranking that places it above the LIBRARY containing the failing version of the program in the search order. This might mean placing the LIBRARY before DFHRPL in the search order.
3. If you did not use EXEC CICS CREATE to define the LIBRARY, install the new LIBRARY resource using CEDA or the CICSPlex SM WUI.
4. Issue a EXEC CICS SET PROGRAM NEWCOPY or EXEC CICS SET PROGRAM PHASEIN command, or the equivalent via CICSPlex SM or CEMT, for the affected program or programs.

Results

The CICS system continues to run and the corrected version of the application is now in the search order before the problem version so that the fixed version is used instead.

Installing a new application to a CICS system

Introduce a new application, which has been provided in one or more data sets, into a running CICS system without affecting continuous availability.

About this task

- The application has been provided in one or more PDS or PDSE data set. The application could be a third party (vendor) product that is provided as a set of application artifacts in one or more PDS or PDSE data set or a new in-house application.
- You cannot restart CICS at this time, to apply the fix.
- The person who performs the action has the appropriate access authorization.

Procedure

1. Use the Business Application Services (BAS) component of CICSPlex SM, or use CEDA or DFHCSDUP or EXEC CICS CREATE, to define a LIBRARY resource, which includes the data set or data sets containing the new application.

Typically, the application will have no intersects with any existing LIBRARY resources and can use the default ranking value.

2. If you did not use EXEC CICS CREATE to define the LIBRARY, install the new LIBRARY resource using CEDA or the WUI.
3. Define to CICS the programs, and map sets that make up the application and transaction definition or definitions that reference it.
4. Install the programs and other definitions.

Results

The new application is installed in the CICS production system and continuous availability is maintained.

Installing a new application to a set of CICS systems

Introduce a new application which has been provided in one or more data sets, into a set of CICS systems in a CICSplex. Such systems are more likely to be in production, although they could also be in a test or a development CICSplex.

About this task

- The application has been provided in one or more PDS or PDSE data sets.
- The application will be introduced to multiple CICS systems at the same time.
- You cannot restart the CICS regions at this time to add the new application or the application is not critical to the running of CICS.
- The person who performs this action has the appropriate access authorization.

Procedure

1. Define a CICSPlex SM LIBRARY definition (LIBDEF), that includes the application data set, using CICSPlex SM BAS.
2. Specify a ranking for the LIBRARY that reflects its ordering relative to other LIBRARY resources in use in the CICS regions. Typically, the application will have no intersects with any existing LIBRARY resources and can use the default ranking value.
3. Install the new LIBDEF, specifying a target scope that covers the set of CICS systems.
4. Define to CICS the programs, map sets and any other artifacts that make up the application and transaction definition or definitions, that reference it.
5. Install the programs and other definitions into the set of CICS systems and start to use them.

Results

The CICS regions are running with the new application.

Reorganizing CICS applications in the LIBRARY

You can reorganize applications in the LIBRARY to make them easier to track and manage. Applications stored in the LIBRARY are defined as DD card data sets in the DFHRPL concatenation.

Before you begin

Ensure that you have the required access authorization.

About this task

The goal of this task is to restructure the organization of applications into LIBRARY resources, so that the data set names are related to the applications they contain, not to operational suitability.

Procedure

1. Decide how to allocation the applications to LIBRARY data sets, and whether to use a single LIBRARY for each application, or multiple applications for each LIBRARY. If you use a single application for each LIBRARY the system configuration is simpler and easier to maintain. You must also determine which applications require multiple data sets to be concatenated in the LIBRARY, and which applications require single data sets.
2. Determine which applications remain in DFHRPL and which applications become dynamic resources.
3. Decide which of the applications to be defined in dynamic LIBRARYs are critical to CICS startup, and which applications are not critical to startup.
4. Define LIBRARY resources for each application, or for each set of applications if they are to be grouped, to be a dynamic resource, using the Business Application Services (BAS) component of CICSPlex SM, CEDA, DFHCSDUP, or EXEC CICS CREATE.
   a. Specify a ranking for each LIBRARY that represents its ordering relative to other libraries in the CICS region. Typically, the application does not intersect with other libraries, and can use the default ranking value.
   b. Specify a status for each LIBRARY that is critical to the running of CICS. Keep the default status, NONCRITICAL, for libraries that are not critical to the running of CICS.
   c. Specify the names of the data sets in the LIBRARY.
5. Install the new LIBRARY resources using either the CEDA INSTALL LIBRARY command, or the CICSPlex SM WUI.
6. Remove those data sets that contain applications in dynamic LIBRARY concatenations, from the DFHRPL concatenation on the next CICS restart.
7. Optionally, reset the ranking for each LIBRARY to its intended permanent value, if you set the ranking to a value that placed the LIBRARY before DFHRPL for testing purposes.
8. Install the new LIBRARY resources during CICS restart using a GRPLIST, during BAS installation, or after a CICS restart. When you do this, the system loads the programs from the new LIBRARY resources because they are now no longer in the DFHRPL concatenation.

Results

CICS runs normally. The LIBRARY applications are now better organized, and are easier to track. It is also easier to determine which CICS systems applications are installed on.

Taking a LIBRARY offline or removing an application from a CICS system

Take a LIBRARY offline; for example to compress a PDS or remove an application from a running CICS system.

About this task

- The application is in a known data set or set of data sets in a dynamic LIBRARY resource.
- The person who performs the action has the appropriate access authorization.

Procedure

1. Disable the LIBRARY using the EXEC CICS SET LIBRARYcommand, CEMT or the WUI.
2. When all uses of the application have completed, perform an operation to cause the loaded copies of the programs to be removed, for example a SET PROGRAM NEWCOPY, or let them fail at the next reload.
3. One reason for disabling the LIBRARY might be to compress the data set and then re-enable the LIBRARY or reinstall the LIBRARY definition to start using the application again. After re-enabling the LIBRARY, issue PROGRAM NEWCOPY or PHASEIN to use the programs again.

Results

While the LIBRARY is disabled, no new users can use the application, unless a copy exists in another LIBRARY that comes after the disabled LIBRARY in the search order, in which case it is loaded from there.

Switching between two LIBRARY concatenations

Introduce one LIBRARY to CICS and take another LIBRARY offline, so that a program in the new LIBRARY is loaded to replace that program in the old LIBRARY.

About this task

- A LIBRARY containing the program or various program artifacts making up an application is currently installed in CICS.
- A new version of the program or application is available in one or more PDS or PDSE data sets.
- The person who performs this action has the appropriate access authorization.

Procedure

1. Define a CICSPlex SM LIBRARY definition (LIBDEF) that includes the new application data set or data sets, using CICSPlex SM BAS.
2. Install the new LIBRARY.
3. Issue a PROGRAM NEWCOPY or PHASEIN command to start using the new copy of the program or programs.
4. Disable the old LIBRARY resource and discard it, if it is not likely to be used again.
5. Optionally, change the ranking of the new LIBRARY using the SET LIBRARY command, the WUI, or the CICSPlex SM API, back to that of the old LIBRARY.

Results

CICS runs with the new LIBRARY and new version of the application.

Discovering information about LIBRARY resources in a CICS system

Discover information about LIBRARY resources.

Before you begin

- CICS is running.

About this task

Discover information about LIBRARY resources, such as the following:

- Which LIBRARY resources are installed in CICS.
- The current search order through the active LIBRARY resources in CICS; for example, installed and enabled LIBRARY resources.
- The relative positions of two LIBRARY concatenations in the search order.
- Which LIBRARY resources are critical.
- The data sets that are defined to a LIBRARY concatenation.

Procedure

If you do not specify a LIBRARY or properties , this inquiry shows all installed LIBRARY resources in the current search order. If you specify some properties of the LIBRARY resources, a subset of installed LIBRARY resources are shown. If a specific LIBRARY is specified, details for that LIBRARY are shown.

Results

The inquiry shows the critical status and enablement status of the LIBRARY resources, their rankings, and also their absolute position in the overall search order. Disabled LIBRARY resources appear in the list but do not participate in the search order. The detailed view of a LIBRARY shows the data sets in its concatenation. If using the CICSPlex SM WUI, click the **Number of DSNAMEs** field name to display the LIBRARY data set names records. You can issue inquiry requests or commands to compare the search position numbers of two LIBRARY concatenations to determine which is before the other in the overall search order.

Discovering LIBRARY information for programs in a CICS system

Inquire on programs in the CICS system to discover the LIBRARY and data set in that LIBRARY from which the program was loaded; for example, to validate that it is loaded from the intended location.

Before you begin

- CICS is running.
- Programs are in use in the CICS system.

Procedure

Use the WUI or CEMT INQUIRE PROGRAM command to inquire on a program.

Results

The program information returned includes the LIBRARY and data set from which it was loaded:

- If the program was loaded from an installed LIBRARY, the LIBRARY and LIBRARYDSN names are returned.
- If the program was loaded from a LIBRARY that has been disabled, the LIBRARY name is returned but the LIBRARYDSN is blank.

- If the program was loaded from a LIBRARY that has been discarded, both LIBRARY and LIBRARYDSN are blank.
- If the program has not been loaded, both LIBRARY and LIBRARYDSN are blank.
- If the program was loaded from the LPA, both LIBRARY and LIBRARYDSN are blank.

Amending the CRITICAL property of LIBRARY resource

Specify that a LIBRARY in one or more CICS regions is critical to CICS startup.

Before you begin

- CICS is running.
- At least one dynamic LIBRARY is active in the CICS system.

Procedure

1. Use the CICSPlex SM WUI, CEMT, or the SPI to change the critical status of an installed LIBRARY. This change takes effect at the next warm or emergency start, when the critical status determines whether or not CICS startup continues uninterrupted if any of the data sets in the LIBRARY concatenation are unavailable or if any other problem prevents the LIBRARY from being recovered as enabled.
2. For a permanent change to the CRITICAL status, use CICSPlex SM BAS or CEDA to define LIBRARY definitions with the required critical status, and install the definitions at each CICS cold or initial start using CICSPlex SM BAS installation or GRPLIST installation.

Results

The CICS behavior on restart changes depending on the critical setting for the LIBRARY and the availability of the data sets in the LIBRARY.

Keeping track of changes to the LIBRARY configuration

Use the audit log to determine changes to the LIBRARY configuration in a CICS system

Before you begin

- CICS is running.

About this task

Use the audit log to determine changes to the LIBRARY configuration in a CICS system, such as the following:

- A new LIBRARY is installed.
- A LIBRARY is removed (discarded) from CICS.
- The ranking, critical status, or enablement status of a LIBRARY is changed.
- The overall LIBRARY search order is changed .

Procedure

1. Examine the audit log, which is written to the CSLB transient data queue, to see changes to a LIBRARY configuration in a CICS system and the resulting LIBRARY search order.
2. Optionally, use a utility, developed in-house or by a vendor, to analyze and interpret the audit log for this system or for multiple systems.

Results

CICS continues running.

Tidying up the LIBRARY configuration

Tidy up LIBRARY concatenations that have been used to apply temporary fixes or that contain application that are no longer used.

Before you begin

- CICS is running.

Procedure

1. Study the names of LIBRARY resources installed in CICS and the audit log of LIBRARY changes, to discover any LIBRARY resources used to apply temporary fixes that are no longer required, or LIBRARY resources for applications that are no longer used.
2. Discard any LIBRARY resources that are no longer required.
3. Delete the definitions of these LIBRARY resources, unless you believe that they will be required in the future and operational procedures allow reuse of LIBRARY definitions in this way.

Results

CICS continues as before. The set of LIBRARY resources installed in the CICS system are only those that are required for current applications used in the system.

Defining MVS residence and addressing modes

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, are described.

About this task

A command-level program can reside above 16 MB, and can address areas above 2 GB.

Establishing the addressing mode of a program

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.

About this task

Generally, your program is designed to execute in that addressing mode, although you can switch modes in the program, and have different AMODE attributes for different entry points in a load module. The RMODE attribute indicates where in virtual storage your program can reside.

Table 9. Valid AMODE and RMODE specifications

Attribute	Meaning
AMODE(24)	Specifies 24-bit addressing mode.
AMODE(31)	Specifies 31-bit addressing mode.

Table 9. Valid AMODE and RMODE specifications (continued)

Attribute	Meaning
AMODE(ANY)	Specifies 24-bit or 31-bit addressing mode.
AMODE(64)	Specifies 64-bit addressing mode.
RMODE(24)	The module must reside in virtual storage below 16 MB. You can specify RMODE(24) for 31-bit or 64-bit programs that have 24-bit dependencies.
RMODE(ANY)	The module can reside in virtual storage below 16 MB, or above 16 MB but below 2 GB.

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.

Procedure

1. You can specify AMODE and RMODE on the link-edit MODE control statement:

   ```
   MODE AMODE(31),RMODE(ANY)
   ```

2. You can specify AMODE and RMODE by using one of the following methods:
   - In the PARM string on the EXEC statement of the link-edit job step:

     ```
     //LKED    EXEC  PGM=IEWL,PARM='AMODE(31),RMODE(ANY),..'
     ```

   - On the LINK TSO command, which causes processing equivalent to that of the EXEC statement in the link-edit step.
3. You can specify AMODE or RMODE statements in the source code of an assembler program. You can also set these modes in COBOL by using the compiler options. See the relevant application programming information for your COBOL compiler.

CICS address space considerations

The valid combinations of the AMODE and RMODE attributes, and their effects, are listed.

Table 10. Valid AMODE and RMODE specifications and their effects

AMODE	RMODE	Residence	Addressing
24	24	Below 16 MB	24-bit mode
31	24	Below 16 MB	31-bit mode
ANY	24	Below 16 MB	31-bit mode
31	ANY	Above 16 MB	31-bit mode
64	24	Below 16 MB	64-bit mode
64	ANY	Above 16 MB	64-bit mode

The following example shows link-edit control statements for a program coded to 31-bit standards, where *anyname* is the name of the load module:

```
//LKED.SYSIN DD *
   MODE AMODE(31),RMODE(ANY)
   NAME    anyname(R)
/*
//
```

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.

About this task

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 CICS dynamic storage areas "CICS dynamic storage areas" in the *CICS Performance 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 that the virtual lookaside facility (VLF) manages. CICS locates a program module from the LLA library directory in storage, rather than searching program directories on DASD. When CICS requests a staged module, LLA gets it from storage without any input/output activity.

Running applications in the link pack area

Programs can reside in the link pack area (LPA) provided that they follow certain requirements.

Assembler language, C, COBOL, or PL/I programs must be read-only and adhere to the requirements 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.)

All programs must be link-edited with the RENT and REFR options.

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.

Running application programs in the read-only DSAs

Programs that are eligible to reside above 16 MB, and are read-only, can reside in the CICS extended read-only DSA (ERDSA). Programs that are *not* eligible to reside above 16 MB, and are read-only, can reside in the CICS read-only DSA (RDSA) below 16 MB.

The following conditions apply for programs to be eligible for the ERDSA:

- Programs must be properly written to read-only standards.
- Programs must be written to 31-bit addressing standards.
- Programs must be link-edited with the RENT attribute and the RMODE(ANY) residency attribute.

The following conditions apply for programs to be eligible for the RDSA:

- Programs must be properly written to read-only standards.
- Programs must be link-edited with the RENT attribute and the RMODE(24) residency attribute.

Note: When you run CICS with `RENTPGM=PROTECT` specified as a system initialization parameter, the RDSAs are allocated from key-0 read-only storage.

ERDSA requirements for the specific languages are described in the following topics.

Assembler

For CICS to load your assembler programs in the ERDSA, assemble and link-edit them with the following options: the RENT assembler option; the link-edit RENT attribute; the RMODE(ANY) residency mode.

If you specify these options, ensure that the program is truly read-only (for example, it does not write to static storage), otherwise storage exceptions occur. The program must also be written to 31-bit or 64-bit addressing standards. For possible causes of storage protection exceptions in programs that are resident in the ERDSA, see Causes of protection exceptions in Troubleshooting.

AMODE(31) programs

The CICS-supplied procedure for AMODE (31) application programs, 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'
```

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), use appropriate statements in your assembler program. For example:

```
MYPROG    CSECT
MYPROG    AMODE 31
MYPROG    RMODE ANY
```

You can set AMODE and RMODE in the following ways:

- 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 data set in the link-edit, or the binder, step in your JCL.

When you use the binder, you might see unexpected warning messages about conflicting AMODE and RMODE specifications.

AMODE(64) programs

The CICS-supplied procedure for AMODE(64) application programs, DFHEGTAL, 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
//EGTAL    EXEC DFHEGTAL,
           .
           (other parameters as necessary)
           .
//         LNKPARM='LIST,XREF,RMODE(ANY),RENT'
```

The CICS EXEC interface module for AMODE(64) assembler programs (DFHEAG) specifies AMODE(64) 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(64) and RMODE(ANY), use appropriate statements in your assembler program. For example:

```
MYPROG    CSECT
MYPROG    AMODE 64
MYPROG    RMODE ANY
```

You can set AMODE and RMODE in the following ways:

- 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 DFHEGTAL,
           LNKPARM='LIST,XREF,RENT,AMODE(64),RMODE(ANY)'
  ```

- Alternatively, you can use the MODE control statement in the SYSIN data set in the link-edit, or the binder, step in your JCL.

When you use the binder, you might 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 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 98.

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'
```

If you want to compile C or C++ code that contains sequence numbers, for example the C or C++ sample programs distributed with CICS, then you must override the **CPARM** parameter to specify SEQ. For example:

```
//EXAMPLE EXEC DFHYITEL,
// CPARM='/CXX OPT(1) SEQ NOMAR SOURCE'
//TRN.SYSIN DD  *
    ... source code goes here ...
/*
//LKED.SYSIN DD *
  NAME EXAMPLE(R)
/*
```

COBOL

If you use the integrated CICS translator, 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, for the following reasons:

- The CBLCARD translator option (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 NOCBLCARD translator option, 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 98.

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 that they do not change static storage.

The following requirements are enforced, either by CICS or PL/I:

- The CICS translator automatically includes the required REENTRANT option 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 98.

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'
```

Do not specify the RENT attribute on the link-edit step unless you have ensured the program is truly read-only (for example, it does not write to static storage), otherwise storage exceptions will occur. For possible causes of storage protection exceptions in programs that are resident in the ERDSA, see Causes of protection exceptions in Troubleshooting.

Using BMS map sets in application programs

The procedure to use BMS map sets in application programs is summarized.

About this task

Use the following procedure before you install an application program to run under CICS.

Procedure

- Create any BMS map sets used by the program, as described in Chapter 10, "Installing map sets and partition sets," on page 149.
- Include the physical map sets (used by BMS in its formatting activities) in a data set that is in the DFHRPL or dynamic LIBRARY 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 CICSTS51.CICS.SDFHMAC library. To include symbolic map sets in a user copy library, use the following steps:

  1. 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.
  2. 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 CICSTS51.CICS.SDFHMAC library (the default), include a DD statement for the CICSTS51.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 CICSTS51.CICS.SDFHMAC library in the SYSLIB concatenation.
  3. 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.

What to do next

For more information about installing map sets, see Chapter 10, "Installing map sets and partition sets," on page 149. For information about writing programs to use BMS services, see Chapter 43, "Basic mapping support," on page 687.

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 supported programming language.

After CICS is installed, copy these procedures, which are installed in the CICSTS51.CICS.SDFHPROC library, into a procedure library. The name of each procedure is in the form DFH*wx*T*y*L, 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. The following tables show the procedure names.

Table 11. Procedures to install application programs: non-Language Environment-conforming compilers

Language	Stand-alone translator	EXCI
Assembler	DFHEITAL (AMODE(24) and AMODE(31) applications) DFHEGTAL (AMODE(64) applications)	DFHEXTAL

Table 12. Procedures for installing application programs: Language Environment-conforming compilers

Language	Stand-alone translator	EXCI	Integrated translator
C	DFHYITDL (see note 1 on page 134)	DFHYXTDL	DFHZITDL (see note 1 on page 134)

Table 12. Procedures for installing application programs: Language Environment-conforming compilers (continued)

Language	Stand-alone translator	EXCI	Integrated translator
C using the XPLINK compiler option	DFHYITFL (see note 2)	-	DFHZITFL (see note 1)
C++	DFHYITEL (see note 1)	DFHYXTEL	DFHZITEL (see note 1)
C++ using the XPLINK compiler option	DFHYITGL (see note 2)	-	DFHZITGL (see note 1)
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 can also be used for C; you must 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. You must use a separate translator step for EXCI COBOL programs, because translator options are ignored when you use the integrated CICS translator.
4. DFHZITCL is the recommended procedure for compiling COBOL modules, because it uses the version of the Enterprise COBOL compiler that includes the integrated CICS translator. However, the integrated translator cannot be used if the COBOL program is intended for batch processing using the EXCI option.
5. DFHZITPL is the recommended procedure for compiling PL/I modules, because it uses the version of the Enterprise PL/I compiler that includes the integrated CICS translator. However, the integrated translator cannot be used if the PL/I program is intended for batch processing using the EXCI option.
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 or dynamic LIBRARY concatenation with primary and secondary extents, and secondary extents are added as a result of link-editing into the load 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

If you want to use CPI Communications or SAA® Resource Recovery in your application program then you must make the appropriate interface modules available to your program.

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(DFHELII).

The supplied procedures for PL/I and C also refer to DFHEILID, which means that the DFHELII 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 can appear during the link-edit step, indicating DUPLICATE definitions for the DFHEI1 entry. You can ignore these messages.

For more information about link-edit requirements, see "Using your own job streams" on page 145.

Translating, assembling, and link-editing assembler language application programs

You can use the DFHEITAL or DFHEXTAL procedure to translate, assemble, and link-edit AMODE(24) and AMODE(31) application programs written in assembler language. You can use the DFHEGTAL procedure to translate, assemble, and link-edit AMODE(64) application programs written in assembler language.

About this task

You can use the sample job control statements shown in Figure 19 on page 136 to process application programs written in assembler language. In the procedure name, *x* depends on whether your programs are CICS application programs and the AMODE of those programs, or EXCI batch programs. For the names of the

CICS-supplied procedures, see "Using the CICS-supplied procedures to install application programs" on page 133.

```
//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)
/*
//
```

Figure 19. Sample job control statements to call the DFHExTAL procedures

anyname is the name of your load module.

Notes:

1. To install a program into a read-only DSA, see "Running application programs in the read-only DSAs" on page 129 for more details.

   To install a program to use from the LPA, add the following options:

   - RENT to the PARM options in the EXEC statement for the ASM step of the DFHE*x*TAL procedure
   - RENT and REFR options to the LNKPARM parameter on the call to the DFHE*x*TAL procedure

   (See "Running applications in the link pack area" on page 128.)
2. For information about the translator options you can include on the XOPTS statement, see "Defining translator options" on page 95.

The following example shows the job control statements to translate, assemble and link-edit an AMODE(64) application program by using the CICS-supplied procedure DFHEGTAL.

```
//APPLPROG EXEC DFHEGTAL
//TRN.SYSIN  DD *
   .
   . Application program
   .
/*
//LKED.SYSIN DD *
   ENTRY program_name
   NAME program_name(R)
/*
```

program_name is the name of the AMODE(64) application.

Figure 20 on page 137 shows the Assembler source program processed by the command level translator, with reference to CICS.SDFHLOAD, 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 to produce a linkage editor listing and a load module that is stored in an application library.

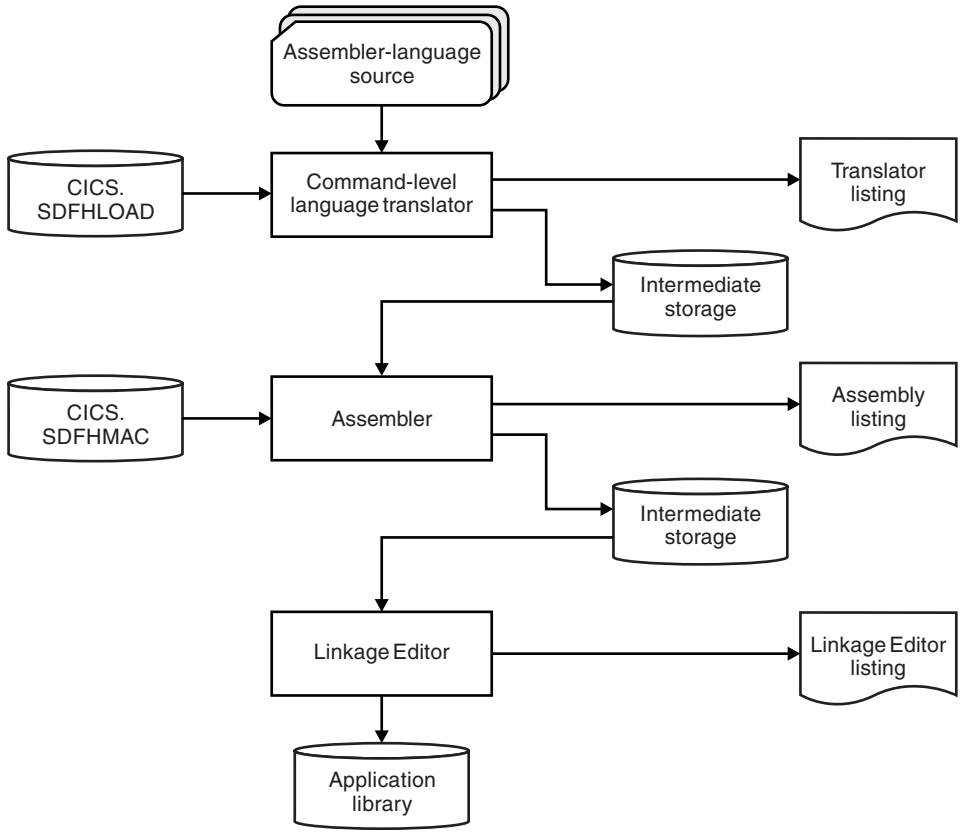


Figure 20. Installing assembler language programs using the DFHEITAL or DFHEGTAL procedure

Installing COBOL application programs

This diagram shows you 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.

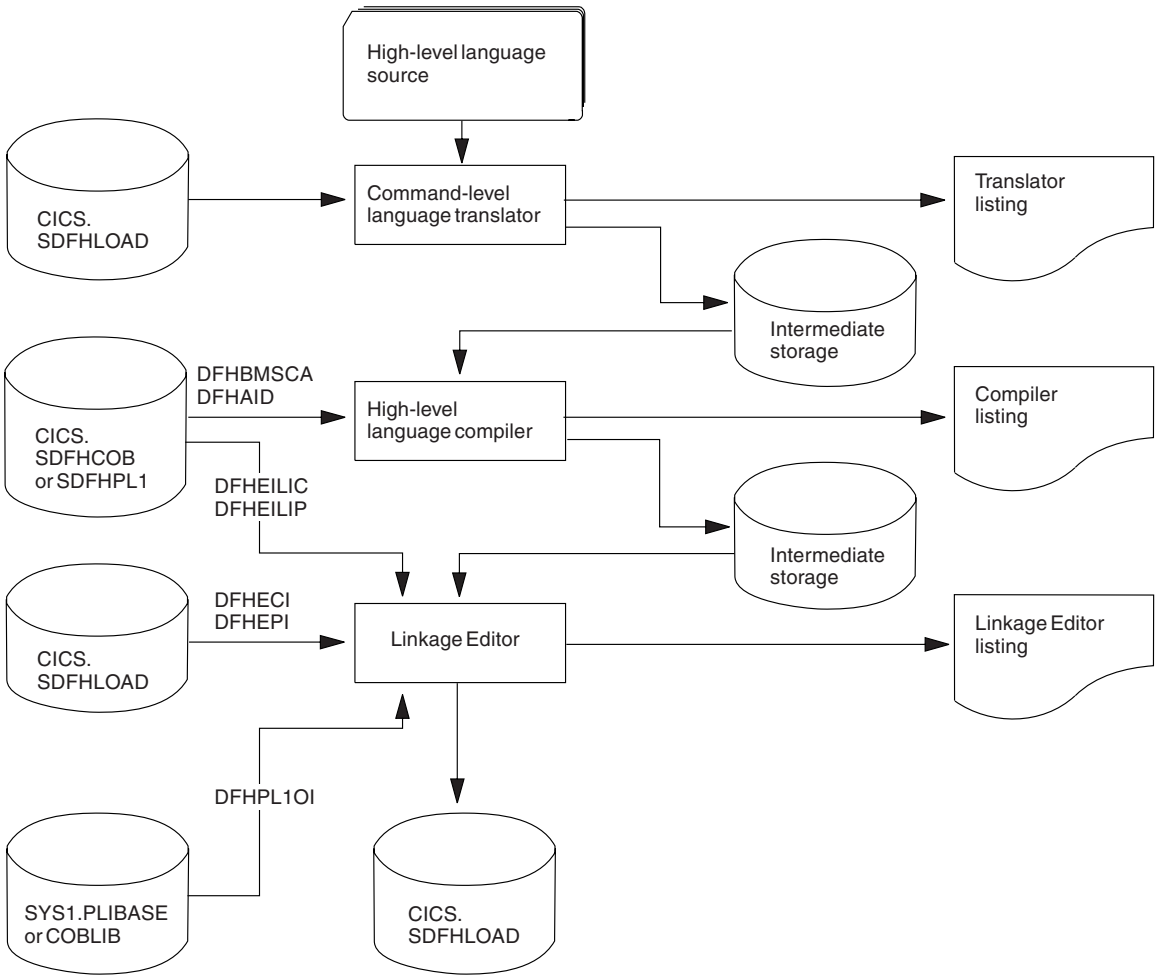


Figure 21. Installing COBOL and PL/I programs

Sample JCL to install COBOL application programs

You can use the job control statements that are shown in these examples to process COBOL application programs with a separate or integrated translator.

The procedure name depends on whether it is a CICS application program or an EXCI batch program. For the names of the COBOL procedures supplied with CCIS, see Table 12 on page 133.

```
//jobname    JOB  accounting info,name,MSGLEVEL=1
//           EXEC PROC=procname                              1
//TRN.SYSIN  DD      *                                      2
 CBL   XOPTS(Translator options . . .)                      3
             .
            COBOL source statements
             .
/*
//LKED.SYSIN DD      *                                      4
              NAME    anyname(R)
/*
//
```

Figure 22. Sample job control statements to call the DFHYITVL or DFHYXTVL procedures

where *procname* is the name of the procedure, and *anyname* is your load module name.

To use the procedure DFHZITCL to invoke the integrated translator, you can use the job control statements that are shown in Figure 23:

```
//jobname    JOB  accounting info,name,MSGLEVEL=1
//           EXEC DFHZITCL,PROGLIB=dsnname                   1
//COBOL.SYSIN  DD *
   .
   . COBOL source statements
   .
/*
//LKED.SYSIN DD *
            NAME anyname(R)
/*
//
```

Figure 23. Sample job control statements to use the DFHZITCL procedure

where *anyname* is your load module name.

Notes:

1. Translator options: specify the COBOL3 or COBOL2 translator option according to the version of the COBOL functionality that is required in the compile step.
2. 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 that contains 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 COBOL procedures supplied with CICS 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 must 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 COBOL procedures supplied with CICS 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,.,.,.'
```

Change compiler options by using any of the following methods:

- Override the PARM statement that is defined on the COB step of the COBOL procedures.

  If you specify a PARM statement in the job that calls the procedure, it overrides all the options that are specified in the procedure JCL. Ensure that all the options you want are specified in the override, or in a CBL statement.
- Specify a CBL statement at the start of the source statements in the job stream that is used to call the COBOL procedures.
- Use the COBOL installation defaults macro, IGYCOPT. This macro is required if you do not use a CBL statement (that is, you specify the translator option NOCBLCARD).
- Define a data set that contains the compiler options for your COBOL program, this data set must include the CICS compiler option and it subparameters.

  Code the SYSOPTF DD statement in one of the following ways:

  ```
  // SYSOPTF  DD  DSNAME=dsname,UNIT=SYSDA,VOLUME=(subparms),DISP=SHR
  ```

  In this code fragment, the compiler options are stored in the data set *dsname*.

  ```
  //COBOL  EXEC PGM=IGYCRCTL,REGION=4M,PARM=(OPTFILE)
  //SYSOPTF  DD *
   APOST
   LIB
   TRUNC(OPT)
   CICS('COBOL3,SP')
   NODYNAM
   RENT
   LIST
   MAP
   XREF
   OPT
   TEST(ALL,SEPARATE)
  //STEPLIB  DD DSN=PP.COBOL390.V410.SIGYCOMP,DISP=SHR
  ```

  In this code fragment, the compiler options are placed directly in the code, after the **OPTFILE** parameter.

  For more information about the SYSOPTF statement, see the Enterprise COBOL for z/OS Information Center:

For information about the translator option CBLCARD|NOCBLCARD, see "Defining translator options" on page 95. If you choose to use the NOCBLCARD option, also specify the COBOL compiler option ALOWCBL=NO to prevent an error message of IGYOS4006-E being issued. For more information about the ALOWCBL compiler option, see the relevant *Installation and Customization* manual for your version of COBOL.

3. 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.
4. If the stand-alone translator supplied with CICS TS is used, the translator options on the XOPTS statement override similar options in the COBOL procedures.

   For information about the translator options you can include on the XOPTS statement, see "Defining translator options" on page 95.

   When the integrated CICS translator is used, the COBOL compiler recognizes only the keyword CICS for defining translator options, not XOPTS.
5. You can ignore weak external references unresolved by the link edit.

   The link-edit job step requires access to the libraries that contain 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 read-only DSAs" on page 129 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 COBOL procedures. For more information, see "Running applications in the link pack area" on page 128.

Installing PL/I application programs

You can use the DFHYxTPL procedures to process PL/I applications with a separate translator. The value of "x" depends on whether it is a CICS application program or an EXCI batch program. The new DFHZITPL procedure can be used to invoke the integrated translator.

Figure 21 on page 138 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*.

Installing C application programs

This diagram shows you the flow of control in the DFHYxTzL cataloged procedures for C command-level 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.

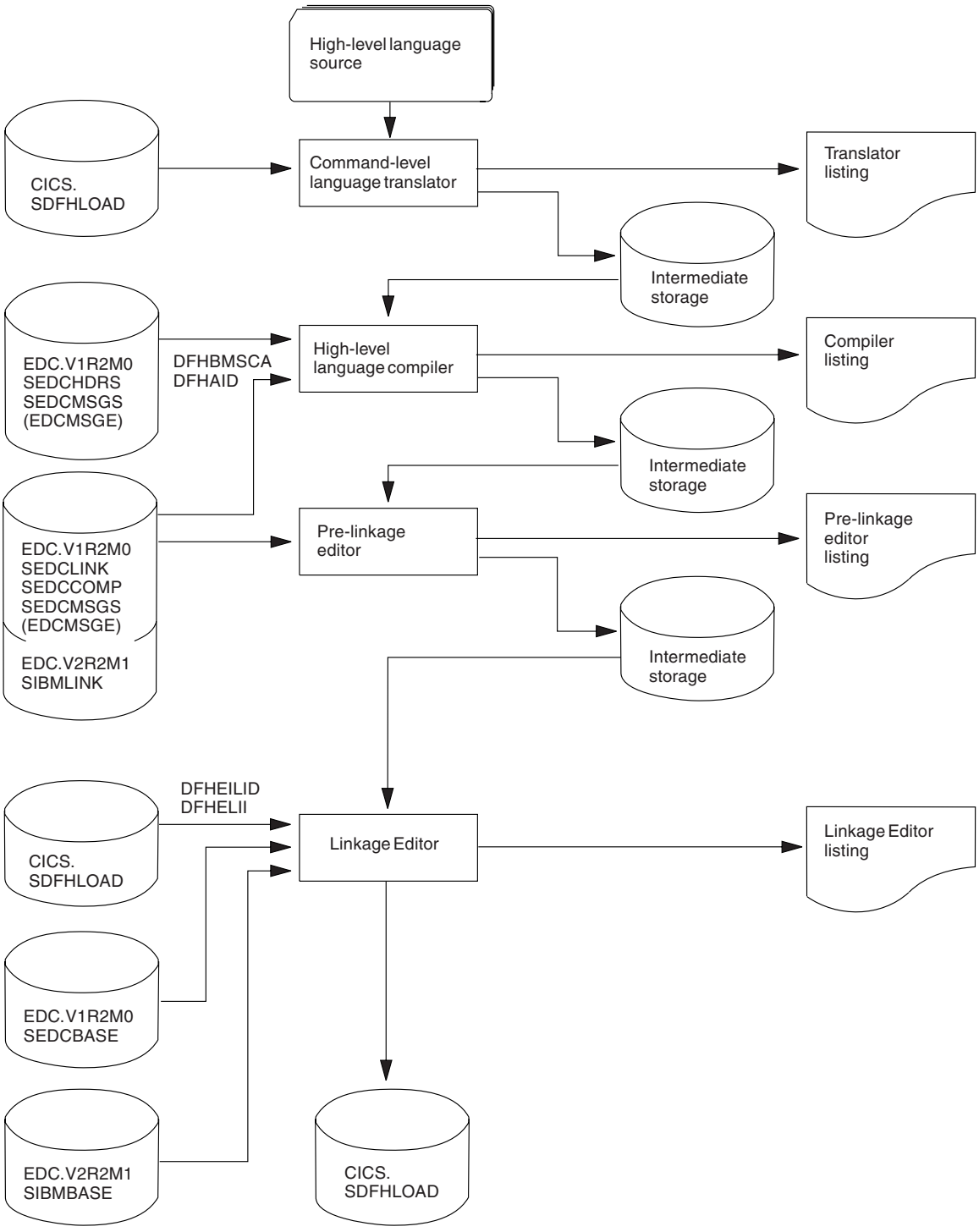


Figure 24. Installing C programs using the DFHYxTzL procedure

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 this diagram.

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 these examples 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 12 on page 133.

```
//jobname    JOB  accounting info,name,MSGLEVEL=1
//           EXEC PROC=DFHYxTzL                                  1
//TRN.SYSIN  DD      *                                          2
#pragma XOPTS(Translator options . . .)                        3
              .
             C source statements
              .
/*
//LKED.SYSIN DD      *                                          4
               NAME    anyname(R)
/*
//
```

where anyname is your load module name.

Figure 25. 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 **CPARM** in the EXEC statement that invokes the procedure, or on a #pragma options directive.

   If you want to make your C or C++ source available to the DFHYITEL, DFHZITEL or DFHYXTEL JCL procedures you must restrict the source input margins. You can restrict the source input margins by:

   - Altering the C or C++ source directly by specifying the following pragma declaration:

     ```
     #pragma margins(m, n)
     ```

     where *m* and *n* are the columns where the C or C++ source is located. For example #pragma margins(1,72).
   - Altering the **CPARM** override options in the JCL procedure; specify SEQ or NOSEQ depending on whether the source has sequence numbers or not.
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 95 .
4. If you are installing a program into either of the read-only DSAs, see "Running application programs in the read-only DSAs" on page 129 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 128 for more information.)

C language programs must be link-edited with AMODE(31), so the DFHYxTzL procedures specify AMODE(31) by default.

Invoking the integrated CICS translator for XL C

To use the procedures to invoke the integrated translator for XL C, you can use the job control statements shown in Figure 26:

```
//jobname JOB accounting info,name,MSGLEVEL=1
// EXEC DFHZITxL,PROGLIB=dsnname                  1
//C.SYSIN DD *
   .
   . C source statements
   .

//LKED.SYSIN DD *
           NAME anyname(R)

//
```

where anyname is your load module name.

Figure 26. Sample JCL for integrated translator for XL C

1. **Translator name:** Specify DFHZITDL for C programs without XPLINK, or DFHZITFL for C programs with XPLINK.

Invoking the integrated CICS translator for XL C++

To use the procedures to invoke the integrated translator for XL C++, you can use the job control statements shown in Figure 27:

```
//jobname JOB accounting info,name,MSGLEVEL=1
// EXEC DFHZITxL,PROGLIB=dsnname                  1
//CPP.SYSIN DD *
   .
   . C++ source statements
   .

//LKED.SYSIN DD *
           NAME anyname(R)

//
```

where anyname is your load module name.

Figure 27. Sample JCL for 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 can generate dfhexec or DFHEXEC. If both versions are present in your program, error message IEW2456E is displayed. There are two ways to prevent this error either recompile the old code containing dfhexec or use prelinker RENAME control statement in the job.

About this task

The following sample JCL shows you how to use the RENAME control statement.

```
//jobname    JOB  accounting info,name,MSGLEVEL=1
//           EXEC PROC=DFHYxTzL
//TRN.SYSIN  DD      *
#pragma XOPTS(Translator options . . .)
             .
            C source statements
             .
/*
//PLKED.SYSLIN DD      *
         RENAME dfhexec DFHEI1
//LKED.SYSLIN DD      *
         NAME anyname(R)
/*
//
```

where anyname is your load module name

Figure 28. Sample JCL to rename dfhexec

Using your own job streams

You can use the supplied cataloged procedures as a model to write your own JCL to translate, assemble (or compile), and link-edit your application programs.

The procedures are installed in the CICSTS51.CICS.SDFHPROC library.

The following information summarizes the important points about the translator and each main category of programs. For simplicity, the descriptions assume that programs are loaded into CICSTS51.CICS.SDFHLOAD or IMS.PGMLIB. You can load programs into any libraries, but only when they are either included in the DFHRPL or dynamic 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, you must rename the IMS libraries accordingly.

Translator requirements

The CICS translator requires a minimum of 256 KB of virtual storage. You might 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 as follows:

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(DFHELII)
```

3. Place the load module output from the link-edit (defined by the SYSLMOD DD statement) in CICSTS51.CICS.SDFHLOAD, or your own program library.

Figure 29 on page 147 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(DFHELII)
```

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 MYYITVL,
//       INDEX='CICSTS51.CICS
//       PROGLIB='CICSTS51.CICS.SDFHLOAD',
//       DSCTLIB='CICSTS51.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 data sets

//TRN.SYSIN  DD *
//*         .
//*         . Application program
//*         .
//*
//LKED.SYSIN DD *
        INCLUDE SYSLIB(DFHELII)
        NAME anyname(R)
//*
//MYYITVL PROC SUFFIX=1$,           Suffix for translator module
//       INDEX='CICSTS51.CICS', Qualifier(s) for CICS libraries
//       PROGLIB='CICSTS51.CICS.SDFHLOAD', Name of o/p library
//       DSCTLIB='CICSTS51.CICS.SDFHCOB',  Private macro/dsect
//       AD370HLQ='SYS1',             Qualifier(s) for AD/Cycle compiler
//       LE370HLQ='SYS1',             Qualifier(s) for Language Environment libraries
//       OUTC=A,                      Class for print output
//       REG=4M,                      Region size for all steps
//       LNKPARM='LIST,XREF',         Link edit parameters
//       WORK=SYSDA                   Unit for work data sets
//*
//*      This procedure contains 3 steps
//*      1.   Exec the COBOL translator (using the supplied suffix 1$)
//*      2.   Exec the COBOL compiler
//*      3.   Linkedit the output into data set &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))
```

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 invoked only 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 CICSTS51.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 CICSTS51.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 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

Use the basic mapping support (BMS) facility of CICS to assemble and link-edit map sets and partition sets. You can use the BMS macros to install HTML templates generated from BMS maps.

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 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 executes.

The process is illustrated in the following diagram:

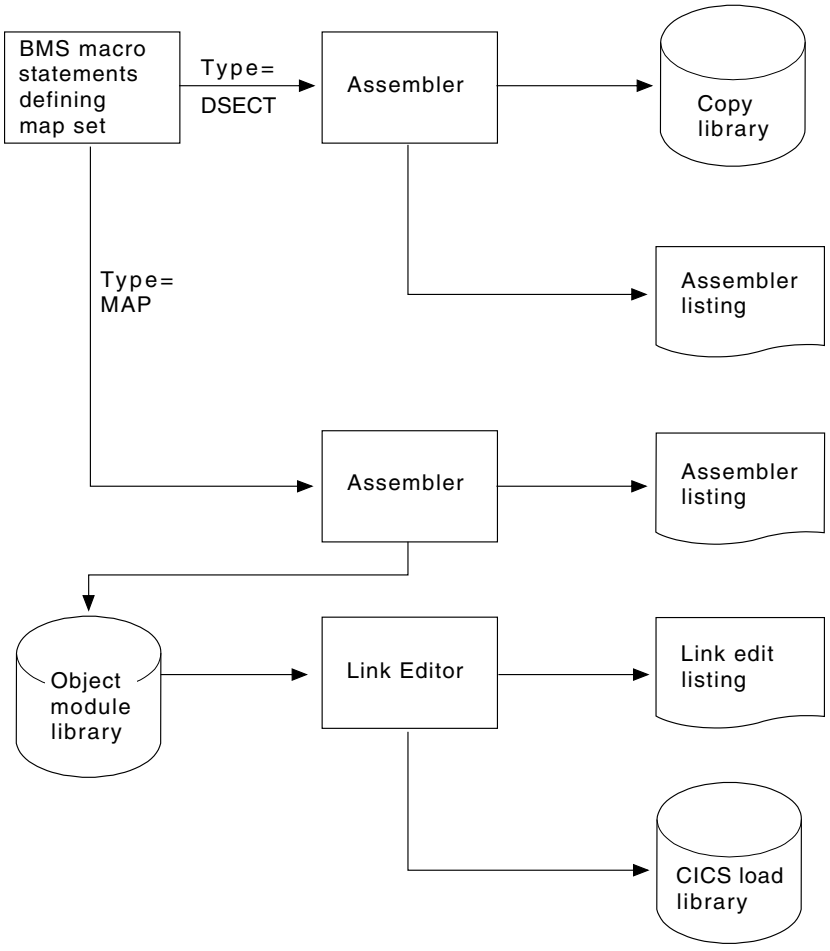


Figure 30. Preparing a map

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 typically need to recompile (reassemble) all programs that use it. Some changes

© Copyright IBM Corp. 1989, 2013

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). For more information about SDF II, see Screen Definition Facility II Library.

CICS loads BMS map sets and partition sets above the 16 MB 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 16 MB 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 section.

This section includes:

- "Installing map sets"
- "Installing partition sets" on page 158
- "Defining programs, map sets, and partition sets to CICS" on page 159

Related concepts:

Chapter 43, "Basic mapping support," on page 687
Basic mapping support (BMS) is an application programming interface between CICS programs and terminal devices.

Related tasks:

"Using the DFHMAPT procedure to install HTML templates from BMS maps" on page 156
The DFHMAPT procedure is similar to DFHMAPS, with an additional step that installs HTML templates generated from the BMS maps.

Installing map sets

Use these examples to learn how to install physical map sets and symbolic description map sets both separately and together.

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.

See:

- "Types of map sets"
- "Installing physical map sets" on page 152
- "Installing symbolic description map sets" on page 154
- "Installing physical and symbolic description maps together" on page 155

Types of map sets

To install a map set, you must prepare two types of map sets; a physical map set and a symbolic description map set.

- A physical map set is 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 is 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

You can distinguish the two types of map set by using either the TYPE operand of the DFHMSD macro or 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 13.

Table 13. 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)

Table 13. SYSPARM and DFHMSD operand combinations for map assembly (continued)

Type of map set	SYSPARM operand of EXEC DFHMAPS statement	TYPE operand of DFHMSD macro
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

You can use fixed extended data stream attributes by reassembling the physical map set or for dynamic attribute medication, by reassembling both the physical, and symbolic description map sets.

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

These examples show you the assembler and linkage editor steps for installing physical map sets and an example job stream.

Figure 31 on page 153 shows the assembler and linkage editor steps for installing physical map sets.

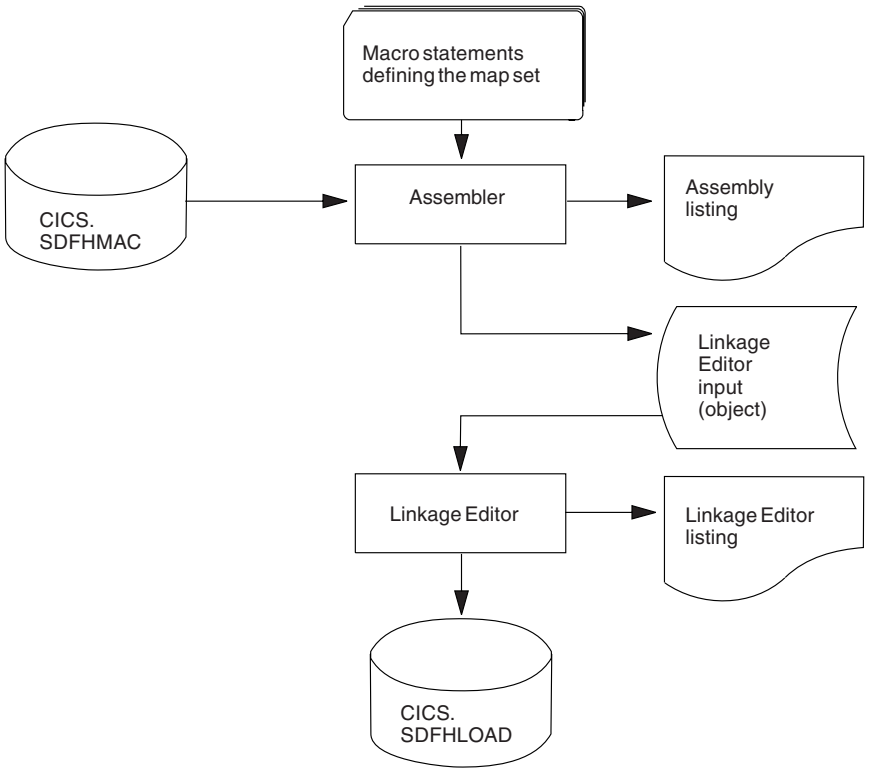


Figure 31. Installing physical map sets

Figure 32 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)'          1
//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'               2
//SYSLIN   DD   DSN=&&TEMP,DISP=(OLD,DELETE)
//         DD   *
           MODE RMODE(ANY|24)                                    3
           NAME mapsetname(R)                                    4
/*
//
```

Figure 32. 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, if 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 Storage protection in Improving performance.

3. The MODE statement specifies whether the map set is to be loaded above (RMODE(ANY)) or below (RMODE(24)) the 16 MB line. RMODE(ANY) indicates that CICS can load the map set anywhere in virtual storage, but tries to load it above the 16 MB 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.

   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 159.

Installing symbolic description map sets

These examples show you the steps for installing symbolic description map sets using the DFHASMVS procedure.

Symbolic description map sets enable the application programmer to make symbolic references to fields in the physical map set. Figure 33 shows the preparation of symbolic description map sets for BMS.

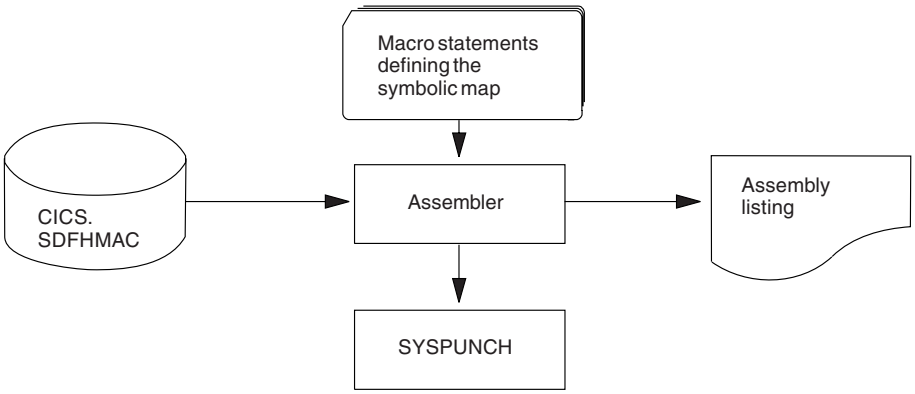


Figure 33. Installing symbolic description map sets using the DFHASMVS procedure

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 34 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 34. 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 34 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 previous 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

These examples show you the steps for installing physical and symbolic description map sets using the DFHMAPS procedure.

Figure 35 on page 156 shows the DFHMAPS procedure for installing physical and symbolic description maps together. The DFHMAPS procedure consists of the following four steps, shown in Figure 35 on page 156:

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 CICSTS51.CICS.SDFHMAC library.

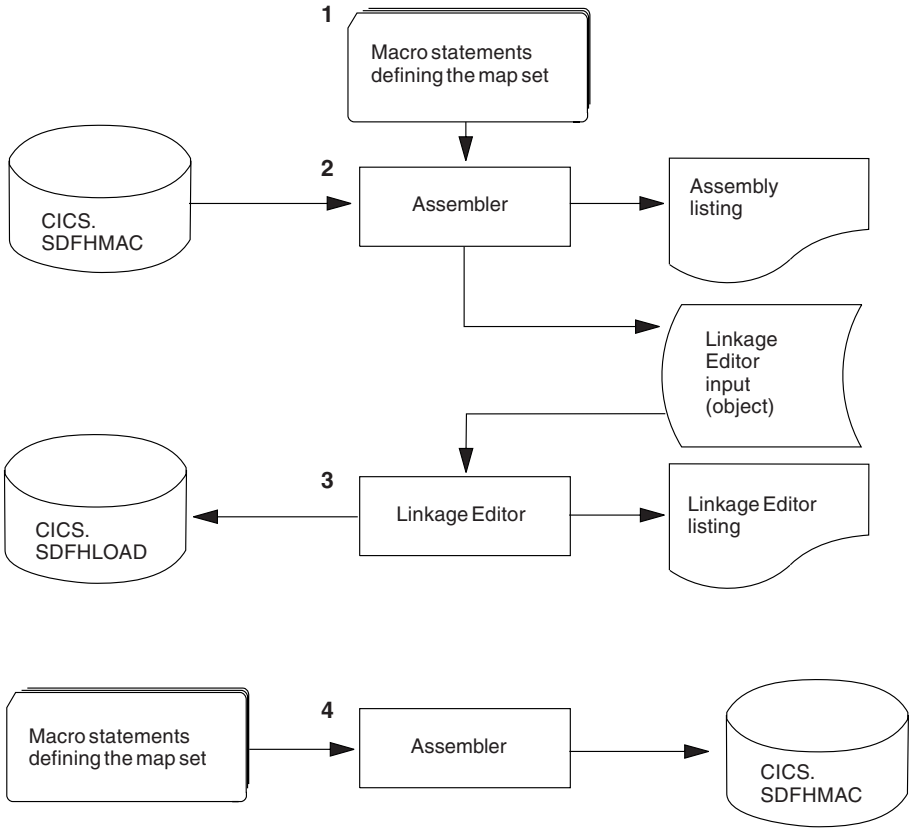


Figure 35. Installing a physical map set and a symbolic description map set together

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.

About this task

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 CICSTS51.CICS.SDFHHTML.

If you want to use your own macro to customize HTML templates, and you do not want 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

This example shows you the JCL job stream needed to install the physical map sets and the symbolic description map sets together.

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 36.

```
//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 36. 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 36, 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 CICSTS51.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

This example shows you how to add both a CSECT name and AMODE and RMODE statements 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 that your maps reside above 16 MB, 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 37. Adding a CSECT to the map assembly

Installing partition sets

You can install partition sets in the same way as physical map sets. There is no concept of a symbolic description partition set.

The job stream in Figure 38 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 38. 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, if RENTPGM=PROTECT is specified on the **RENTPGM** system initialization parameter.

   For more information about the storage protection facilities available in CICS, see Storage protection in Improving performance.

2. The MODE statement specifies whether the partition set is to be loaded above (RMODE(ANY)) or below (RMODE(24)) the 16 MB 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 Reference*.

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).

About this task

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*.

Chapter 11. Testing applications

You can use the following methods to test CICS application programs. This guidance does not relate to testing Java applications.

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 the basic CICS information (such as resource definition) is correct. It is 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 might find that a transaction that works perfectly in its single-thread testing still fails in the multithread test. It might 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 Chapter 29, "Using sequential terminal support," on page 543), 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 can 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

© Copyright IBM Corp. 1989, 2013

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

This list shows you what you need to consider when preparing the application and system table entries.

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. Create a PROFILE resource definition for your transactions to use, and make sure that the definition is installed.
5. Create a TRANSACTION resource definition for each transaction in your application, and make sure that the definitions are installed.
6. If your system does not use program autoinstall, create a PROGRAM resource definition for each program used in the application, and make sure that the definitions are installed.
7. If your system does not use program autoinstall, create a MAPSET resource definition for each map set in the application, and make sure that each definition is installed.
8. Create a FILE resource definition for each file used, and make sure that each definition is installed.
9. Build at least a test version of each of the files required.
10. Define each of the transient data queues 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

This list shows you what you need to consider when preparing the system for debugging.

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 DFHDU680 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 typically a workable value.
7. Generate statistics. For more information about using statistics, see CICS statistics in Monitoring.

Chapter 12. 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 section 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

© Copyright IBM Corp. 1989, 2013

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 168
- "Testing programs using EDF" on page 175
- "Using EDF to change information" on page 180
- "Using EDF menu functions" on page 182

Restrictions when using EDF

When you are using EDF to debug your application programs, you must be aware of a number of restrictions.

Open TCBs and EDF

Even if your program typically runs 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. 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 might overlay the parameter list for EXEC CICS commands issued by the main program.

Security considerations

EDF is such a powerful tool that your installation might restrict its use with attach-time security. The external security manager used by your installation defines the security attributes for the EDF transaction. If you are not authorized to use CEDF, you cannot initiate the transaction.

Application prerequisites

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 already 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).

Restrictions for single-screen mode

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, the user transaction and CEDF must not 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, do not specify PROTECT=YES in the profile definition of the CEDF transaction. 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.
- To test an application program that uses screen partitions, or that does its own request unit (RU) chaining, 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 ends with the ADIR abend. 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, certain commands (for example, ASSIGN and ADDRESS) are issued against the EDF terminal and not the transaction terminal. See INVOKE CECI for information about how to invoke CECI from CEDF.

- When using EDF in dual-screen mode, 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 might 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, might cause EDF to appear as if it has finished. If the task ends abnormally, EDF is reactivated at the monitoring terminal.

Restrictions for both screen modes

Other restrictions apply to both screen modes:

- If a transaction issues the FREE command, EDF is switched off without warning.
- 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 or password phrase value to reduce the risk of accidental disclosure.

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 39 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
F1 : UNDEFINED          F2 : SWITCH HEX/CHAR  F3 : END EDF SESSION
F4 : SUPPRESS DISPLAYS  F5 : WORKING STORAGE  F6 : USER DISPLAY
F7 : SCROLL BACK        F8 : SCROLL FORWARD   F9 : STOP CONDITIONS
F10: PREVIOUS DISPLAY   F11: EIB DISPLAY      F12: ABEND USER TASK
```

Figure 39. Typical EDF display

Note: 1Header 2Body 3Message line 4Menu 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 function keys F7 and F8. The header, menu, and message areas are repeated on each screen.

The header

The header of the display contains the following information; 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, and the 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 you the body contents at program initiation, at the start and the end of execution of a CICS command, at program and task termination, and at abnormal termination.

At program initiation

This example shows you the display at program initiation. 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*. If there isn't a COMMAREA, line 4 on the screen remains 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
    EIBAID       = X'7D'                              AT X'032F059A'
    EIBFN        = X'0000'                            AT X'032F059B'
    EIBRCODE     = X'000000000000'                    AT X'032F059D'
    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 40. Typical EDF display at program initiation

At the start of execution of a CICS command

This example shows you the display at the start of execution of a CICS command. EDF displays the command, including keywords, options, and argument values.

A typical EDF display at start of execution of a CICS command is shown in Figure 41 on page 170. 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 41. Typical EDF display at start of execution of a CICS command

Figure 42 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 42. 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 41. (See "Defining translator options" on page 95 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

This example shows you the display at 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 41 on page 170 is shown in Figure 43.

```
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 43. 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 the *CICS Application Programming Reference*, and DL/I codes are documented in the *Application Programming: EXEC DLI Commands*.

Figure 44 on page 172 and Figure 45 on page 172 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 44. 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 45. 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 46. Typical SQL display at completion of an SQL command

At program and task termination

This example shows you the display at program termination and normal task termination. There is no body information; all the pertinent information is in the header.

Figure 47 and Figure 48 on page 174 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 47. 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 48. Typical EDF display at task termination

At abnormal termination

This example shows you the display when an abend or abnormal task termination occurs.

When an **abend** or **abnormal task termination** occurs, EDF displays the screens shown in Figure 49 and Figure 50 on page 175.

```
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
   EIBAID       = X'7D'                          AT X'032F059A'
   EIBFN        = X'1804'  SEND                  AT X'032F059B'
   EIBRCODE     = X'000000000000'                AT X'032F059D'
   EIBDS        = '........'
+  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 49. 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 50. 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

Testing programs using EDF

You can run EDF by using either the CEDF or CEDX transaction. If you are testing a nonterminal 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, use the CEDF transaction.

You can typically select one of several methods for running EDF, but there are a few situations in which a specific method is required. For example, you must use single-screen mode for remote transactions. See “Restrictions when using EDF” on page 166 for other conditions that affect your choice.

Interrupting program execution

Using EDF, you can perform different operations at intercept points in your program to help debug it.

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 command itself, 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 “Using EDF to change information” on page 180 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 EXEC interface block (EIB), and for DL/I programs, the DL/I interface block (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.
- Stop the task with an abend.

The first two types of changes are made by overtyping values in the body of the command displays. "Using EDF to change information" on page 180 tells you how to do this. You use the function keys in the menu for the others; "Using EDF menu functions" on page 182 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 51. 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 can start EDF in two ways:

- Entering transaction code CEDF from a cleared screen
- Pressing the appropriate function key (if one has been defined for EDF)

Next, you start the transaction to be tested by completing the following steps:

1. Press the CLEAR key to clear the screen.
2. Enter 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 the 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 you use CEDF with an application program that has been translated with option NOEDF, or one that has NO specified for CEDF in its resource definition, EDF cannot ascertain when the display is changed by that application program. Therefore EDF cannot save a copy of that display for later use. The next EDF display overwrites any display sent by the application program and cannot be restored.

Similarly, 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. Therefore an output command to the principal facility from the application program might result in random background information from a previous EDF display appearing on the screen.

An input command can 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 might require explicit input from the CEDF panel instead of being satisfied by the contents of the initial TIOA.

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, 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, 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.

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 to accept the default, which 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 that is being tested.

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 transaction 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
```

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 might be looping, you can go to another terminal and enter a CEDF transaction that names 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 that is being tested, or the terminal that invokes it, is owned by another CICS region.

If a remote transaction ends abnormally 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

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 nonterminal transactions, use the CEDX *trnx* command, where *trnx* is the transaction identifier.

To test a transaction using CEDX, the following conditions must be met:

- 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 test a transaction that is using distributed transaction processing (DTP) across a remote link by telling execution diagnostic facility (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 have 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 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 that the transaction is using:

```
CEDF sessionid
```

You can determine the session identifier with the **INQUIRE TERMINAL** command, but this means that the transaction must be running and must have reached the point of establishing a session before you start EDF.

If a transaction that uses 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 usual way from that terminal.

When you have finished testing the transaction on the remote system, turn off EDF on that system identifier or session identifier before logging off from CICS with CESF. For example:

```
CEDF sysid,OFF
```

Failure to turn off EDF could cause another transaction that is 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 system identifier 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 program status word (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 that is being tested 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, overtype 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.

Using EDF to change information

Most of the changes you make with EDF involve changing information in memory. To make these changes, type over the information that is displayed on the screen with the information that you require.

You can change any area that you can move the cursor to by using the tab keys, except for the menu area at the bottom of the screen.

When you change the screen, you must observe the following rules:

- On CICS command screens, you can type over any argument value, but not the keyword of the argument. You cannot remove an optional argument, and you cannot add or delete an option.

- When you change an argument in the command display (rather than the working storage screen), you can change only the part shown on the display. If you attempt to type 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, change the area in working storage to which the argument points. To determine the address, display the argument in hexadecimal format so that the address of the argument location is also displayed.
- When you type over an argument value that is a fullword, the maximum value that you can enter is 2147483639.
- In character format, numeric values always have a sign field. You can type over a sign field with only a minus character (-) or a blank.
- When an argument is to be displayed in character format, some characters might not be displayable (including lowercase characters). EDF replaces each nondisplayable character with a period. If you type over a period, remember that the storage might contain a nondisplayable character.

  You cannot type over a character with a period to change it; the change is ignored and no diagnostic message is issued. To change a character to a period, switch the display to hexadecimal format, use the F2 key, and type over with the value `X'4B'`.
- When storage is displayed in both character and hexadecimal format, if you change both formats and the changes conflict, the value of the hexadecimal field takes precedence. No diagnostic message is issued.
- The arguments for some commands, for example, 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 and you cannot modify it, for example, ERROR ( ).
  - 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').
  - For AMODE(64) assembler language, labels are not supported.

  If no label value is specified on a HANDLE CONDITION command, EDF displays the condition name alone without the parentheses.
- You can type over the response field with the name of any exception condition that can occur for the current function, including ERROR, or the word NORMAL. The effect when EDF continues is that the program takes the action that is 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 uppercase translation is not specified for the terminal you are using, ensure that you always enter uppercase characters.
- You can type over any command with NOOP or NOP before processing to suppress processing of the command. You can type over with blanks or use the ERASE EOF key for the same effect. When the screen is redisplayed with NOOP,

you can restore the original verb line by erasing the whole verb line with the ERASE EOF key and pressing the ENTER key.

- You can enter 64-bit addresses if the argument is already a 64-bit address, including addresses with an underscore in the middle, for example AAAAAAAABBBBBBBB or AAAAAAAA_BBBBBBBB.

When you type over a field that represents 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 change might 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, you might test a program that contains the following code:

```
EXEC CICS SEND MAP('MENU') END-EXEC.
```

If you use EDF to change the name from MENU to MENU2 before you run the command, the map used is MENU2, but the map displayed on the response is MENU. You can use the "previous display" key to verify that 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

You can use this list of rules to learn the EDF responses to any keyboard entry.

The rules are as follows, in the order shown:

1. If the CLEAR key is used, EDF redisplays the screen with any changes ignored.
2. If invalid changes are made, EDF accepts any valid changes and redisplays 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 function key is used, EDF accepts any changes and performs the action requested by the function key. Pressing ENTER with the cursor under a function key definition in the menu at the bottom of the screen is the same as pressing a function key.
5. If the ENTER key is pressed and the screen has been modified (other than the REPLY field), EDF redisplays 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 redisplays the status display.

Using EDF menu functions

The function keys that you can use at each point are displayed in a menu at the bottom of every EDF display.

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. 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. The message `ENTER ABEND CODE AND REQUEST ABEND AGAIN` is displayed to confirm this action. Enter the abend code at the cursor position, then 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 that start with the character A are reserved for use by CICS. Using a CICS abend code might 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 CEBR*xxxx*, where *xxxx* is the terminal identifier of the terminal running EDF. This function is only available from the working storage (PF5) screen. You can use CEBR commands to display or modify temporary storage queues and to read or write transient data queues.

CONTINUE
Redisplays the current screen to incorporate any changes. If you made no changes, CONTINUE causes the transaction under test to resume execution up to the next intercept point. To continue, press ENTER.

CURRENT DISPLAY
Redisplays the current screen to incorporate any changes. If you made no changes, EDF displays 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 *IMS: Application programming for EXEC DLI* for information about DIB fields.

EIB DISPLAY
Displays the contents of the EXEC interface block (EIB). For an example of an EIB display, see Figure 40 on page 169. For programming information about the EIB, see EIB fields in Reference -> Application development. If COMMAREA exists, EDF also displays its address and one line of data in the dump format.

INVOKE CECI
Accesses the command-level interpreter (CECI). This function is only available from the working storage (PF5) screen. See Figure 51 on page 176 for an example of the screen from which CECI is invoked. You can then use CECI commands, discussed in Chapter 14, “Command-level interpreter (CECI),” on page 199. 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 for restrictions when running CECI in dual-screen mode. The use of CECI from this panel is like 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
If you returned to a previous display, displays the next one forward and increases the display number by one. This option is the reverse of PREVIOUS DISPLAY.

PREVIOUS DISPLAY

Sends the previous display to the screen, 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 is no longer available on the menu, and the corresponding function key is inoperative.

REGISTERS AT ABEND

Displays storage that contains the values of the registers if a local ASRA abend occurs. The layout of the storage is 64-bit registers, followed by 16-byte program status word (PSW) at abend. Figure 52 shows a typical screen.

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 situation occurs, a second test run might provide more information.

```
TRANSACTION: UT   PROGRAM: ASRA31   TASK: 0000487 APPLID: IYK2ZKE1 DISPLAY:  00
  ADDRESS: 0006F010
 0006F010   000000   00000000 00000090 00000000 0010003C   ................      1
 0006F020   000010   00000000 00041800 00000000 AE410028   ................      1
 0006F030   000020   00000000 2C2E35A8 00000000 00000000   .......y........      1
 0006F040   000030   00000000 2C2D7800 00000000 7F2C9918   ............".r.      1
 0006F050   000040   00000000 2E410000 00000000 2C2D7818   ................      1
 0006F060   000050   00000000 00100008 00000000 00100100   ................      1
 0006F070   000060   00000000 009AF000 00000000 00100690   ......0.........      1
 0006F080   000070   00000000 AE410158 00000000 00100690   ................      1
 0006F090   000080   079D0000 80000000 00000000 2E410178   ................      2
 0006F0A0   000090   00040004 00000000 00000000 00000000   ................
 0006F0B0   0000A0   00000000 00000000 00000000 00000000   ................
 0006F0C0   0000B0   00000000 00000000 00000000 00000000   ................
 0006F0D0   0000C0   00000000 00000000 00000000 00000000   ................
 0006F0E0   0000D0   00000000 00000000 D4F0F0F0 F0F4F8F9   ........M0000489
 0006F0F0   0000E0   D4F0F0F0 F0F4F8F9 00000000 00000000   M0000489........
 0006F100   0000F0   00000000 000000E4 E3404000 00F00000   .......UT  ..0..

 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 52. Typical EDF display for REGISTERS AT ABEND

Notes:

1. Register values
2. PSW

REMEMBER DISPLAY

Places a display that would not typically 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

View the previous screen in the display. This function 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, select this function to view previous screens in the display. See Figure 40 on page 169 for an example.

SCROLL FORWARD
View the next screen in the display. This function applies to an EIB, DIB, or command display that does not all fit on one screen. When the display does not fit, a plus sign (+) appears after the last option or field in the display, to show that there are more screens. Select this function to display the next screen.

SCROLL BACK FULL
View the previous screen in a working-storage display. This function applies to displays of working storage and is similar to 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 addresses on the current screen.

SCROLL FORWARD FULL
View the next screen in a working-storage display. This function applies to displays of working storage and is similar to 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 addresses on the current screen.

SCROLL BACK HALF
Reverse the display of working storage by half a screen. This function is similar to SCROLL BACK FULL, except that the display of working storage is reversed by only half a screen.

SCROLL FORWARD HALF
Advance the display of working storage by half a screen. This function is similar to SCROLL FORWARD FULL, except that the display of working storage is advanced by only half a screen.

STOP CONDITIONS
Specify which conditions cause EDF to resume displays after using the SUPPRESS DISPLAYS function. The menu screen shown in Figure 53 on page 186 is displayed. You can use the STOP CONDITIONS and SUPPRESS DISPLAYS functions together to reduce the interaction when you check 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 53. Typical EDF display for STOP CONDITIONS

By default, EDF resumes displays if any of the following conditions occur, as shown in Figure 53:

- Any CICS exception condition
- Transaction abend
- Normal task termination
- Abnormal task termination

You can use the STOP CONDITIONS menu to turn off any default that does not apply, and add conditions specific to your program.

You can specify any or all 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 that the program was translated with the DEBUG option) is encountered.
- Any DL/I error status occurs, or a specific DL/I error status occurs.
- A specific exception condition occurs. When ERROR (the default) is specified for CICS EXCEPTION CONDITION, EDF redisplays a screen in response to any error condition (for example, NOTOPEN, EOF, or INVREQ). If you specify a specific condition, such as EOF, for CICS EXCEPTION CONDITION, and NO (the default) is specified for ANY CICS CONDITION, EDF redisplays the screen only when that condition (EOF) occurs.

  If you specify YES for ANY CICS CONDITION, EDF overrides the CICS exception conditions and redisplays 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.

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 specify that EDF resumes displays at DL/I commands as well as at CICS commands. Type over the CICS qualifier on the command line with `DLI` and enter the type of DL/I command at which you want display suppression to stop. You must be running 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 display 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 *IMS: Application programming for EXEC DLI*.

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 ten previous command displays, even though they were not 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 does not affect previously remembered displays, STOP CONDITIONS displays, or working storage displays.

In DL/I command displays that contain the WHERE option, only the key values (the expressions that follow each comparison operator) can be converted to hexadecimal.

UNDEFINED
The indicated function key is not defined for the current display at the current intercept point.

USER DISPLAY
Displays 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
Shows the contents of the 24-bit or 31-bit working storage area in your program or of any other address in the CICS region. Figure 54 on page 188 shows a typical working storage screen.

| This function does not support 64-bit storage.

```
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 54. Typical EDF display for working storage

The working storage contents are displayed in a form like 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 enter a new address at the top of the screen. This address can be anywhere in 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; for example, it might not be required if the program does not issue CICS commands. When you LINK to such a program, the following message might be issued: `Register 13 does not address DFHEISTG`. This 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 72 byte standard format save area; that is, registers 14 to 12 begin at offset 12, and register 13 is stored at offset 4. For AMODE(64) programs, working storage starts with a format 4 save area (F4SA); registers 14 to 12 begin at offset 8, and register 13 is stored at offset 128.

Working storage can be changed at the screen; ou can use either the hexadecimal section or the character section. You can type over the ADDRESS field at the head of the display with a hexadecimal address; storage starting at that address is then displayed when you press ENTER. You can examine any location in the address space. For more information, see “Using EDF to change information” on page 180.

If you are examining program storage that is not part of the working storage of the program currently running, which is unique to the particular transaction under test, the corresponding field on the screen is protected to prevent you 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 situation 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 you can still examine any storage area in the region by typing over the address field.

If you terminate a PL/I or Language Environment program with an ordinary non-CICS return, EDF does not intercept the return, and you cannot 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 13. 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 can 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 192
- "Using the CEBR function keys" on page 193
- "Using the CEBR commands" on page 194
- "Using the CEBR transaction with transient data" on page 197

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 enter a name using as many as 16 characters. For example, to display the temporary storage queue named AXBYQUEUENAME111 you type `CEBR AXBYQUEUENAME111` and press ENTER. If the queue name includes lowercase characters, ensure that uppercase translation is suppressed for the terminal you are using, and then enter the correct combination of upper and lowercase characters. CICS responds with a display of the queue, for example, as shown in Figure 55 on page 192.

Alternatively, you can start the CEBR transaction from the CEDF transaction. You do this by pressing PF5 from the initial CEDF screen (see Figure 39 on page 168) 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 56 on page 192. If you enter the CEBR transaction from the CEDF transaction, you return to the EDF panel when you press PF3 from the CEBR screen.

© Copyright IBM Corp. 1989, 2013

```
 CEBR  TSQ AXBYQUEUENAME111 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 55. Typical CEBR display of temporary storage queue contents

```
 CEBR  TSQ AXBYQUEUEAME1AA  SYSID CIJP REC     1 OF     0    COL     1 OF    1
 ENTER COMMAND ===>                                                          2
      *************************  TOP OF QUEUE  ******************************
      ************************  BOTTOM OF QUEUE  ****************************









                                                                             3
TS QUEUE AXBYQUEUEAME1AA        DOES NOT EXIST                               4
PF1 : HELP                PF2 : SWITCH HEX/CHAR     PF3 : TERMINATE BROWSE  5
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
```

1. Header
2. Command area
3. Body
4. Message line
5. Menu of options

Figure 56. Typical CEBR display of default temporary storage queue

What does the CEBR transaction display?

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 55 on page 192 and (AXBYQUEUEAME1AA in Figure 56 on page 192). You can overtype this field in the header if you want to switch the screen to another queue. If the queue name includes lowercase characters, ensure that uppercase translation is suppressed for the terminal you are using, and then enter the correct combination of upper and lowercase 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 overtype 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 194. 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."

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.

For example, the "Does not exist" message shown in Figure 56 on page 192.

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, and 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

CEBR provides a number of 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 197 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 197 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 lowercase characters, ensure that uppercase translation is suppressed for the terminal you are using, and then enter the correct combination of upper and lowercase 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 typically 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 are 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 can 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 14. 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.

CECI 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 can 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 205
- "Saving commands" on page 207
- "How CECI runs" on page 208

What does CECI display?

All CECI screens have the same basic layout. CECI displays consist of the command line, the status line, the screen body, the message line, and the CECI option on function keys.

- "The command line"
- "The status line" on page 200
- "The screen body" on page 203
- "The message line" on page 204
- "CECI options on function keys" on page 204

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.

© Copyright IBM Corp. 1989, 2013

- 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

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.

Command syntax check

When the status line shows **command syntax check**, 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 example shows you the typical CECI display for about to execute command.

This display (as shown in Figure 57 on page 202) 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 57. Typical CECI display for about to execute command

If you press the ENTER key without changing the screen, CECI executes the command. You can still modify it, 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.

CICS command completes

This example shows you the typical CECI display for command execution complete.

This display (as shown in Figure 58 on page 203) appears after the interpreter has run 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( +0000000041 ) >
   < BAsedsname( '                                          ' ) >
   < BLOCKFormat( +0000000016 ) >
   < BLOCKKeylen( -0000000001 ) >
   < BLOCKSize( -0000000001 ) >
   < BRowse( +0000000039 ) >
   < Cfdtpool( '        ' )
   < DElete( +0000000043 ) >
   < DIsposition( +0000000027 ) >
   < DSname( 'CFV01.CICS03.PSK.CSD                       ' ) >
   < EMptystatus( +0000000032 ) >
   < ENAblestatus( +0000000033 ) >
   < EXclusive( +0000000001 ) >
   < Fwdrecstatus( +0000000361 ) >
+  < Journalnum( +00000 ) >


  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 58. 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 screen body

The body of the CECI screens contains information that is 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" on page 204. 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.

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 action produces an expanded display.

If the command has more options than can fit on one screen, a plus sign (+) appears at the left side of the last option of the current display to indicate that there are more. An example of this is shown in "CICS command completes" on page 202. 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.

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 might 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.

Figure 58 on page 203 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 58 on page 203) 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 function keys for the display.

The function keys are described below. If the terminal has no function keys, the same effect can be obtained by positioning the cursor under the required item in the menu and pressing ENTER.

F1 HELP
displays a HELP panel giving more information on how to use the command interpreter and on the meanings of the function keys.

F2 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.

F3 END
(END SESSION) ends the current session of the interpreter.

F4 EIB
(EIB DISPLAY) shows the contents of the EXEC interface block (EIB).

F5 VAR
(VARIABLES) shows all the variables associated with the current command interpreter session, giving the name, length, and value of each.

F6 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.

F7 SBH
(SCROLL BACK HALF) scrolls the body half a screen backward.

F8 SFH
(SCROLL FORWARD HALF) scrolls the body half a screen forward.

F9 MSG
(DISPLAY MESSAGES) shows all the messages generated during the syntax check of a command.

F10 SB
(SCROLL BACK) scrolls the body one full screen backward.

F11 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. 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 command summary in Reference -> Application development and the *CICS System Programming Reference*.

If you use the transaction code CECS, the interpreter checks your command for correct syntax. If you use CECI, you have the option of executing your command when 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.

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 overtype a period, the current value might not be a period, but might be 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.

Defining variables

When the command might exceed the line length of the command input area because of the values of the options, or you want to connect two commands through option values, you can define variables.

To display the list of variables, press PF5. For each variable associated with the current interpreter session, the name, length, and value are displayed. CECI creates the first three variables and they are always displayed, unless you explicitly delete them. They are intended to provide examples and help you create command lists.

After these first three variables, any variables that you have created are displayed. For example, you might enter the following command:

```
READ FILE('FILEA') RID('009000') INTO(&REC)
```

The `&REC` is displayed as a variable.

Typically, the value supplied for an option in the command line is taken as a character string constant. However, you can specify a variable to represent this value, for example when you want to connect two commands through option values. For example, to change a record with CECI, you might first enter the following command:

```
EXEC CICS READ UPDATE INTO(&REC)
          FILE('FILEA') RID('009000')
```

You can then modify the record by changing the variable &REC, and then enter the following:

```
EXEC CICS REWRITE FROM(&REC) FILE('FILEA')
```

The ampersand in the first position tells CECI that you are specifying a variable.

You can create variables with the required values and specify the variable names in the command to overcome the line length limitation. Variables can have a data type of character, doubleword, fullword, halfword, or packed decimal. You can create variables in any of the following ways:

- Name the variable in a receiver. The variable is created when the command is processed. The data type and length are implied by the option.
- Add new entries to the list of variables already defined. To create a new variable, type its name and length in the appropriate columns on the first unused line of the variables display, then press ENTER. For character variables, use the length with which the variable has been defined. For doublewords, type **FD**. For fullwords, type **F**. For halfwords, type **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. After a variable is created, you can change the value by modifying the data field on the **variables** display.

- Use the NAME field on the status line when you have produced an expanded area display of a specific option. To do this, position the cursor under the option on a syntax display and press ENTER. To assign the variable name you want associated with the displayed option value, type it into the NAME field and press ENTER again.
- Copy an existing variable. To do this, obtain an expanded area display of the variable to copy, type over the name displayed with the name of the new variable, and press ENTER.
- Use 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."

You can also delete a variable, although CECI discards all variables at session end. To delete a variable before session end, position the cursor under the ampersand that starts the name, press ERASE EOF, then press ENTER.

Saving commands

Sometimes you might 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 typical 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. If you select the variables display, you 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 " CItttt", 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 the commands you saved.

You can vary this procedure to suit your needs. For example, you can skip commands in the sequence by 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

The interpreter runs as a conversational transaction, using programs supplied by CICS. Everything you do between the start of a session and the end is a single logical unit of work in a single task.

Locks and enqueues produced by commands that 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 runs 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, make sure that you can complete all the updates. If you cannot complete all the updates, use 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 might return exception conditions even when all specified options are correct. These conditions return 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 CECI might return the information you requested correctly, it attempts 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 might produce different results from an application program executing those commands. For example, an **ABEND** command is intercepted unless you use the CANCEL option.

If you use a **LINK** command, the target program runs in the environment of the interpreter. In particular, if you modify a user display during a linked-to program, the interpreter is not aware of the changes.

Similarly, if you interpret an **XCTL** command, CECI passes control to the named program and never gets control back, so the CECI session is ended.

Terminal sharing

When the command that is 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 in the following situations:

- When the command that is being processed requires data from the operator
- When the command that is being processed is about to modify the user display
- When USER DISPLAY is requested

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. You do not have to respond to the SEND command, but if you do, the interpreter stores and displays 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.

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 15. Preparing to use debuggers with CICS applications

CICS supports the use of workstation-based and host-based debuggers for isolating and fixing bugs, and for testing applications. Before you can use a debugger with CICS applications, you must perform the following tasks.

Before you begin

About this task

Procedure

1. 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 16, “Debugging CICS applications from a workstation,” on page 217.

   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 17, “Using Debug Tool with CICS applications,” on page 219.

   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.
2. 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 might 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

© Copyright IBM Corp. 1989, 2013

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."

3. 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 *Java Applications in CICS* for more information.
4. 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.

Example

What to do next

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 z/OS Communications Server netname.
- The name of the program
- For COBOL programs, the name of the compile unit (the program or class name)
- For Java objects, the class 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:
  A3270 terminal
  Adebugging 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

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 14 is an example which shows how parameters in the debugging profiles are used to select program instances for compiled language programs; Table 15 shows how parameters in a debugging profile are used to select the program instance for a Java program.

Table 14. 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 15. 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.

About this task

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 59 on page 216 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 59. The debugging profile matching algorithm

Chapter 16. 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.

About this task

You must complete the following tasks:

Procedure

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.

© Copyright IBM Corp. 1989, 2013

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 Debugging an application that is running in a CICS JVM in *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.

Results

When you have completed all these steps, the programs that you have selected in the final step will run under the control of a debugger.

Chapter 17. Using Debug Tool with CICS applications

Debug Tool helps you test programs and examine, monitor, and control the execution of application programs.

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.

Before you begin

See the *CICS Application Programming Guide* for more information.

© Copyright IBM Corp. 1989, 2013

About this task

You must then complete the following tasks:

Procedure

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.

Results

When you have completed all these steps, the programs that you have selected in the final step will run under the control of Debug Tool.

Part 4. CICS application programming techniques

Concepts and techniques which you need to know to create programs for CICS.

© Copyright IBM Corp. 1989, 2013

Chapter 18. Application design

Use these 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 “Design for performance” on page 252.

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

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

© Copyright IBM Corp. 1989, 2013

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.

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 "Sharing data across transactions" on page 261 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 Recovery and restart facilities in 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 enqueuing, or you can have the transaction compare the original version of the record with the current version before applying the update.

Terminal interruptibility

When a conversational task is running, CICS allows no other messages to be sent to that terminal. Pseudoconversational applications can allow messages to come through between message pairs of a conversation.

Conversational tasks have 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 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 failure of the user to terminate the conversation can prevent or delay a normal CICS shutdown.

Pseudoconversational applications can allow messages, so 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. Prevent this 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 - from unsolicited input, or 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 524), and the SEND PAGE command (in “The SEND PAGE command” on page 736) all do this.

- CICS creates a task to process a transient data queue (see "Automatic transaction initiation (ATI)" on page 599).
- CICS creates a task to deliver a message sent by a BMS ROUTE request (see "Message routing" on page 750). 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 667.

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?

On receipt of an unsolicited input, how does CICS determine which transaction to run? Typically the previous task with the same principal facility determines what transaction CICS runs next, using the TRANSID option on its final RETURN.

This situation is almost always the case in a pseudoconversational transaction sequence, and typically 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 60, a diagram of this logic.

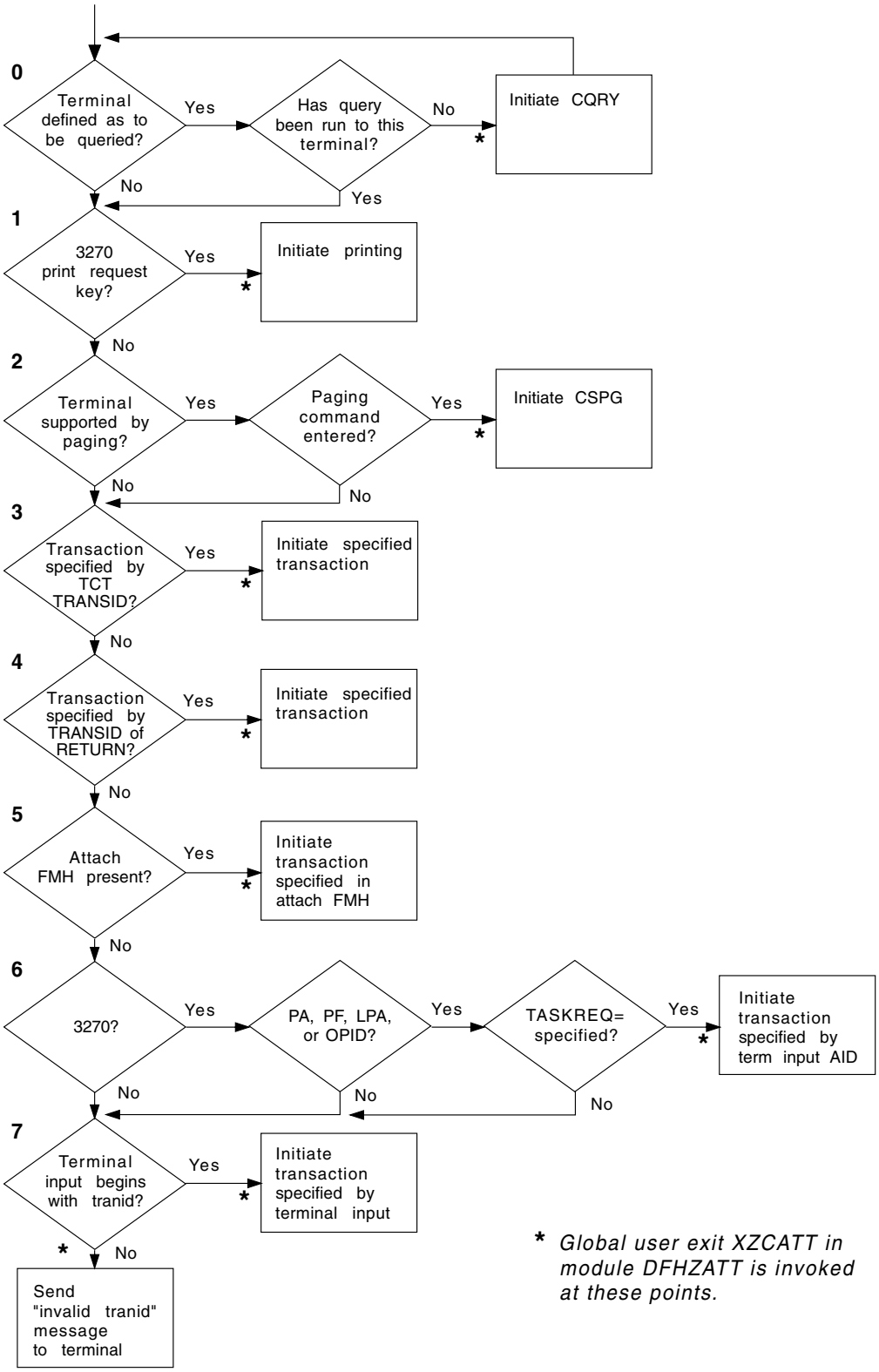


Figure 60. Determining which transaction to execute

0. On the first input from a terminal, CICS sometimes schedules a preliminary task before creating one to process the input. This task runs

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 in "Printing display screens" on page 673 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 687, 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 four-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 in "Input from a 3270 terminal" on page 562 explains attention identifiers. You define one as a transaction identifier with the TASKREQ attribute of the corresponding TRANSACTION definition.
7. If all 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 61 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 61. 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 62 and Figure 63 on page 230illustrate this approach to application design.

```
..
EXEC CICS RECEIVE MAP
..
EXEC CICS LINK..
..
EXEC CICS SEND MAP
..
```

Figure 62. Presentation logic

```
..
EXEC CICS ADDRESS COMMAREA
..
EXEC CICS READ UPDATE
..
EXEC CICS REWRITE
..
EXEC CICS RETURN..
```

Figure 63. 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)" on page 371 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 where a single copy of an application program can be processed by several transactions concurrently. For example, one transaction can begin to execute an application program. When an **EXEC CICS** command is reached, causing a CICS WAIT and call to the dispatcher, another transaction can then execute the same copy of the application program.

Compare this technique 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 are quasi-reentrant; that is, they must be serially reusable between entry and exit points. CICS application programs that use the EXEC CICS interface obey this rule automatically. For COBOL, C, and C++ programs, reentrancy is ensured by obtaining a fresh copy of working storage 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 that 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 set on the PROGRAM resource definition. The following table shows you the CONCURRENCY and API settings that are available on the program, and the type of TCB that the application program is run on.

Table 16. Combination of PROGRAM CONCURRENCY and API settings and the type of TCB used.

CONCURRENCY attribute	API attribute	CICS TCB
CONCURRENCY(QUASIRENT)	API(CICSAPI)	The application program always runs under the CICS quasi-reentrant (QR TCB). If a switch to an open TCB occurs for a resource manager, CICS always returns to QR TCB before returning to the application.
CONCURRENCY(QUASIRENT)	API(OPENAPI)	Invalid combination. OPENAPI programs cannot run on a QR TCB.

Table 16. Combination of PROGRAM CONCURRENCY and API settings and the type of TCB used. (continued)

CONCURRENCY attribute	API attribute	CICS TCB
CONCURRENCY(THREADSAFE)	API(CICSAPI)	The application program can run on a QR TCB or an open TCB. If a switch to an open TCB occurs for a resource manager, CICS stays on the open TCB when returning to the application. If a switch to the QR TCB occurs, CICS stays on QR TCB when returning to the application.
CONCURRENCY(THREADSAFE)	API(OPENAPI)	Same as CONCURRENCY(REQUIRED) API(OPENAPI).
CONCURRENCY(REQUIRED)	API(CICSAPI)	The application program always runs on an open TCB. CICS services do not require TCB key matching so the application always runs on a L8 open TCB. An OPENAPI resource manager uses an L8 TCB so no TCB switch is required if it is invoked. If a switch to the QR TCB occurs, CICS returns to the open TCB when returning to the application.
CONCURRENCY(REQUIRED)	API(OPENAPI)	The application program always runs on an open TCB. The key of the TCB must match the execution key of the program. CICS uses an L9 TCB if EXECKEY(USER) is set and an L8 TCB if EXECKEY(CICS) is set. If the application is user key and an OPENAPI resource manager is invoked then a switch occurs from the L9 TCB to the L8 TCB. CICS returns to L9 TCB before returning to the application. If a switch to the QR TCB occurs, CICS returns to the open TCB when returning to the application.

Quasi-reentrant application programs

A *quasi-reentrant* program is a program that is 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 multithreading through the program by multiple CICS tasks.

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 run concurrently on multiple TCBs.

CICS requires that an application program is reentrant so that it guarantees consistent conditions. In practice, an application program might not be truly reentrant; CICS expects "quasi-reentrancy".

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 running 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 be running.

To ensure that programs cannot interfere with each other's working storage, CICS obtains a separate copy of working storage every time an application program runs. For example, if a user application program is being used 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. For example, an application can update a field in the CWA without using compare and swap (CS) instructions or locking (enqueuing 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 run 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 excluded. However, you can use the task-control SUSPEND command to allow other transaction processing to proceed; see Chapter 34, "Task control," on page 573 for details. 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 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 in the open transaction environment (OTE).

Accessing the open transaction environment

Applications that involve a task-related user exit (TRUE) enabled using the OPENAPI option on the **ENABLE PROGRAM** command are automatically involved with the open transaction environment. These applications can gain performance benefits from being threadsafe. The CICS DB2 task-related user exit used by CICS applications that access DB2 resources is an open API TRUE, so CICS DB2 applications can gain performance benefits from being threadsafe. For more details about threadsafe programming for CICS DB2 applications, see Enabling CICS DB2 applications to use the open transaction environment (OTE) through threadsafe programming in the DB2 Guide.

For other user application programs, PLT programs, user replaceable modules, or task-related user exits, you can opt to use the open transaction environment by defining them as OPENAPI programs, in which case they must be threadsafe. For more details about threadsafe programming for OPENAPI programs, see OPENAPI programs.

If you define your program as CONCURRENCY(REQUIRED) it always runs on an open TCB. The type of open TCB used depends on the API setting. For CICSAPI programs, CICS uses an L8 open TCB regardless of the execution key of the program. For OPENAPI programs, CICS uses an L9 TCB if EXECKEY(USER) is set and an L8 TCB if EXECKEY(CICS) is set. REQUIRED is applicable to user application programs, PLT programs, and user replaceable modules. For global user exit programs and task-related user exit programs, if you specify CONCURRENCY(REQUIRED), CICS treats the program as if you had specified CONCURRENCY(THREADSAFE).

Global user exits cannot be defined as OPENAPI programs, but if you use the THREADSAFE option on the **ENABLE PROGRAM** command for a global user exit, it is enabled as a threadsafe program and can run on the same open TCB as an application that uses it. If the **ENABLE PROGRAM** command does not specify the CONCURRENCY or API options then the options on the program definition are used.

Serialization techniques

An application that is running on an open TCB 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.

For most resources, such as files, transient data queues, temporary storage queues, and DB2 tables, CICS processing automatically ensures access in a threadsafe manner. Some of the CICS commands that operate on these resources are coded to use appropriate serialization techniques that allow them to run 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 behavior of the command.

For any other resources that are accessed directly by user programs, such as shared storage, it is the responsibility of the user program to ensure threadsafe processing. For information about threadsafe programming for user application programs, see Making applications threadsafe.

TCB switching

Threadsafe programs achieve performance benefits by remaining on an open TCB, rather than switching between the open TCB and the QR TCB. When a program is defined as threadsafe and is running on an open TCB, TCB switching from the open TCB to the QR TCB occurs in the following circumstances:

- If the program issues any EXEC CICS commands that are not threadsafe, CICS switches back from the open TCB to the QR TCB to run the nonthreadsafe code. If the program is defined as OPENAPI or CONCURRENCY(REQUIRED), CICS then switches back to the open TCB to continue running the application logic. If

the program is not defined as OPENAPI or CONCURRENCY(REQUIRED), it continues to run on the QR TCB. For a CICS DB2 application, if the program is not defined as OPENAPI or CONCURRENCY(REQUIRED) and does not make any further DB2 requests, the switch back to the QR TCB is 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.

- If the program calls a task-related user exit program that 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 situation is the same as after a nonthreadsafe EXEC CICS command: an OPENAPI or CONCURRENCY(REQUIRED) program switches back to the open TCB, and a program not defined as OPENAPI or CONCURRENCY(REQUIRED) continues to run on the QR TCB.
- When the program issues a threadsafe CICS command or makes a DB2 request, a global user exit might be invoked as part of the processing for the command or request. If a global user exit program is used that 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 or to complete the DB2 request.
- When the program completes, CICS switches back to the QR TCB for task termination. This switch is always necessary.

Threadsafe considerations for statically or dynamically called routines

If you define a program with CONCURRENCY(THREADSAFE) or CONCURRENCY(REQUIRED), all routines that 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 link-edit 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) or CONCURRENCY(REQUIRED), the called routine must also comply with this specification. Programs with the CONCURRENCY(THREADSAFE) or CONCURRENCY(REQUIRED) 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 must determine whether the program can remain

on the open TCB, or whether the program must 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) or CONCURRENCY(REQUIRED), then processing can continue on the open TCB. In this scenario program B is executing, so if processing is to continue safely, program B must be coded to threadsafe standards.

Making applications threadsafe

When you make an application program threadsafe, you can use the open transaction environment, avoid TCB switching, and gain performance benefits.

Before you begin

To use threadsafe application programs, ensure that the system initialization parameter **FORCEQR** is not set to YES. **FORCEQR** forces programs defined as threadsafe to run on the QR TCB, and it might be set to YES as a temporary measure while problems connected with threadsafe-defined programs are investigated and resolved.

Also, select an appropriate setting for the system initialization parameter **FCQRONLY** in your file-owning CICS regions. If **FCQRONLY** is set to YES, CICS forces all file control requests in the CICS region to run on the QR TCB.

- If you use IPIC connections to function ship file control requests to remote CICS TS 4.2 or later regions, to improve performance for those connections set **FCQRONLY** to NO in the file-owning regions.
- If you use only MRO links or ISC over SNA connections, or your file-owning regions are earlier than CICS TS 4.2, set **FCQRONLY** to YES in the file-owning regions.

If you are using CICS intercommunication to make requests for functions or programs to run in remote CICS systems, choose IP interconnectivity (IPIC) over TCP/IP connections between the CICS systems to provide optimal support for threadsafe applications. With IPIC connections, CICS uses an open TCB to run the mirror program that manages the request on the remote CICS system, providing improved throughput. With other connection types, CICS does not use an open TCB to run the mirror program. The EXEC CICS LINK command, used for distributed program link (DPL), is threadsafe for IPIC connections to remote CICS regions where a long-running mirror is used, but not for other connection types.

About this task

Threadsafe programs explains what it means for a program to be threadsafe, and the circumstances when TCB switching takes place between open TCBs and the QR TCB.

To make an application program threadsafe and enable it to remain on an open TCB, use the following procedure.

Procedure

1. Define the program to CICS as threadsafe, by specifying CONCURRENCY(THREADSAFE) in the PROGRAM resource definition.

   For a program that is defined as OPENAPI, CICS requires the CONCURRENCY(THREADSAFE) option. Only code that has been defined as

threadsafe is permitted to run on open TCBs. By defining a program to CICS as threadsafe, you are specifying only 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, but, to permit your program to run on an open TCB, CICS needs you to guarantee that your application logic is threadsafe.

Alternatively, you can define the program as CONCURRENCY(REQUIRED) to enable your program to run from the start on an open TCB. Programs defined as CONCURRENCY(REQUIRED) must be coded to threadsafe standards as they must always run on an open TCB. The type of open TCB used depends on the API setting.

2. Ensure that the program logic, that is, the native language code between the EXEC CICS commands, is threadsafe.

   If you define a program to CICS as threadsafe but include application logic that is not threadsafe, the results are unpredictable, and CICS cannot protect your program from the possible consequences. To make your program logic threadsafe, you must use appropriate serialization techniques when accessing shared resources, to prohibit concurrent access to those resources. When you use EXEC CICS commands to access resources such as files, transient data queues, temporary storage queues, and DB2 tables, CICS ensures threadsafe processing, but for any resources that are accessed directly by user programs, such as shared storage, the user program must 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. 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

   The load module scanner utility includes a sample table, DFHEIDTH, to help you identify CICS commands that give access to shared storage. 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 can potentially not be threadsafe. To ensure that it is threadsafe, an application program must include the necessary synchronization logic to guard against concurrent update.

   Tip: When identifying programs that use shared resources, also include any program that modifies itself. Such a program is effectively sharing storage and you must consider it at risk.

   You can use the following techniques to provide threadsafe processing when accessing a shared resource:

   - 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 one of the following techniques:
     - 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 that can run under the QR TCB might result in performance degradation because of 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.
- 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 approach effectively provides a very coarse locking mechanism, but might 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 be considered threadsafe only 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.

3. For best performance, ensure that the program uses only threadsafe EXEC CICS commands.

   If you include an EXEC CICS command that is not threadsafe in a program that is defined as threadsafe and 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 results of your application are not affected, but its performance might be affected.

   The commands that are threadsafe are indicated in the command descriptions of the CICS API and SPI command topics with the statement “This command is threadsafe”. They are also listed in Threadsafe commands in Reference -> Application development and Threadsafe SPI commands in Reference > System programming.

   The load module scanner utility includes a sample table, DFHEIDNT, to help identify any CICS commands in your applications that are not threadsafe.

4. If any user exit programs are in the execution path used by the program, for best performance ensure that they are also coded to threadsafe standards and defined to CICS as threadsafe. These exits might be dynamic plan exits, global user exits, or task-related user exits. Also check that user exit programs supplied by any vendor software are coded to threadsafe standards and defined to CICS as threadsafe.

   A threadsafe user exit program can be used on the same open TCB as a threadsafe application that calls it, and it can use non-CICS APIs without having to create and manage subtask TCBs, and exploit the open transaction environment for itself. If any user exit programs in the execution path used by the program are not threadsafe, CICS switches to the QR TCB to run them, which might be detrimental to the application's performance.

   Be aware of the following important user exits:

   - The global user exits XEIIN and XEIOUT are called before and after EXEC CICS commands.
   - The global user exit XPCFTCH is called before a program defined to CICS receives control.

- For CICS DB2 requests, the CICS DB2 task-related user exit DFHD2EX1 is threadsafe. Other important exits for CICS DB2 requests include the default dynamic plan exit DSNCUEXT, which is not defined as threadsafe, the alternative dynamic plan exit DFHD2PXT, which is defined as threadsafe, and the global user exits XRMIIN and XRMIOUT.
- If you function ship CICS file control, transient data, or temporary storage requests to a remote CICS region over an IPIC connection, the requests are threadsafe and can run in the remote CICS region on an open TCB. Any global user exit programs that are called in the remote CICS region for file control, transient data, or temporary storage requests must be enabled as threadsafe programs for the best performance.

a. To define a user exit program to CICS as threadsafe, you can specify appropriate attributes in its PROGRAM resource definition.
   - For a task-related user exit program, specify OPENAPI and THREADSAFE, or just THREADSAFE.
   - For a global user exit program, you cannot use OPENAPI, but you can specify THREADSAFE.

   If you specify CONCURRENCY(REQUIRED) on a global user exit program or task-related user exit program, CICS treats the program as if you had specified CONCURRENCY(THREADSAFE).

b. Alternatively, to define a user exit program to CICS as threadsafe, you can specify appropriate options when you enable the program by using the **EXEC CICS ENABLE PROGRAM** command.
   - For a task-related user exit program, specify OPENAPI or THREADSAFE.
   - For a global user exit program, you cannot use OPENAPI, but you can specify THREADSAFE.

   When you enable an exit program using the OPENAPI or THREADSAFE option, this action indicates to CICS that the program logic is threadsafe, so CICS overrides the CONCURRENCY setting on the program definition for the exit and treats the exit program as threadsafe.

c. To define a first-phase PLT global user exit program as threadsafe, specify THREADSAFE on the **EXEC CICS ENABLE PROGRAM** command. To ensure that global user exit programs (such as those that run at the recovery exit points) are available as early as possible during CICS initialization, it is common practice to enable them from first-phase PLT programs. Because first-phase PLT programs run so early in CICS initialization, you cannot use installed PROGRAM resource definitions or the program autoinstall user program to define the exit programs. CICS automatically installs exit programs that are enabled from first-phase PLT programs with CONCURRENCY(QUASIRENT). However, the setting on the **EXEC CICS ENABLE PROGRAM** command overrides the CONCURRENCY(QUASIRENT) setting on the system-autoinstalled program definition.

CONCURRENCY(REQUIRED) programs

Defining an application program as CONCURRENCY(REQUIRED) means that from the start of the program, it always runs on an open task control block (open TCB) instead of the main CICS quasi-reentrant TCB (QR TCB). If CICS must switch to the QR TCB to process an **EXEC CICS** command, CICS switches back to the open TCB before returning control to the application program.

The type of open TCB used depends on what APIs the program is to use:

- If the program uses only CICS supported APIs (including access to external resource managers such as DB2, IMS, and WebSphere MQ), it must be defined with program attribute API(CICSAPI). In this case CICS always uses an L8 open TCB, irrespective of the execution key of the program, because CICS commands do not rely on the key of the TCB.
- If the program is to use other non-CICS APIs, it must be defined with program attribute API(OPENAPI). In this case CICS uses an L9 TCB or an L8 TCB depending on the execution key of the program. This is to allow the non-CICS APIs to operate correctly.

Global user exits cannot be defined as CONCURRENCY(REQUIRED), but if you enable them using the THREADSAFE option on the ENABLE PROGRAM command, they can run on an open TCB when necessary.

Task-related user exits (TRUEs) can be defined or enabled as CONCURRENCY(REQUIRED), in which case, they must be written to threadsafe standards. A TRUE that is defined as CONCURRENCY(REQUIRED) API(OPENAPI) always runs on an L8 TCB. A TRUE that is defined as CONCURRENCY(REQUIRED) API(CICSAPI) can run on a T8, L8, or X8 TCB, depending on the need of the application environment. The CICS-DB2 TRUE is enabled as CONCURRENCY(REQUIRED) API(CICSAPI).

Existing threadsafe CICS-DB2 applications that are defined as THREADSAFE CICSAPI (to take advantage of the performance gains of being able to run on the same TCB as the DB2 call), can be further enhanced by defining them as REQUIRED CICSAPI. This definition means that the programs can run on an L8 open TCB immediately, without waiting for the first DB2 call to move them on to the open TCB. The additional benefit achieved depends on how many, if any, non-threadsafe CICS commands the application executes.

OPENAPI programs

The open transaction environment (OTE) is an environment where CICS application code can use non-CICS services inside the CICS address space, without interference from other transactions. You can use the OTE by defining user application programs, PLT programs, user replaceable modules, or task-related user exits (TRUEs) as OPENAPI programs, by using the OPENAPI attribute in the PROGRAM resource definitions.

Defining a program as an OPENAPI program by using the attribute API(OPENAPI) means that from the start of the program, it always runs on an L8 or L9 mode open task control block (open TCB) instead of the main CICS quasi-reentrant TCB (QR TCB). Global user exits cannot be defined as OPENAPI programs, but if you enable them using the THREADSAFE option on the **ENABLE PROGRAM** command, they can be run on an open TCB when necessary.

Moving application workloads off the QR TCB onto multiple open TCBs gives the possibility of achieving better throughput, particularly with CPU-intensive programs. You can use other non-CICS APIs, but you must note that the use of non-CICS APIs within CICS is entirely at the discretion and risk of the user. No testing of non-CICS APIs within CICS has been undertaken and the use of such APIs is not supported by IBM Service.

OPENAPI programs must be *threadsafe* in order to run on an open TCB. The requirements for a threadsafe program are as follows:

1. The program must be defined to CICS as CONCURRENCY(REQUIRED) meaning that the program is required to run on an open TCB.
2. The logic of the program, that is, the native language code between the EXEC CICS commands, must be threadsafe. If you define a program to CICS to run on an open TCB but include application logic that is not threadsafe, the results are unpredictable, and CICS is not able to protect you from the possible consequences.
3. For best performance, the program must use only threadsafe EXEC CICS commands. If you include a non-threadsafe EXEC CICS command in a program that 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 might be detrimental to the performance of the application.
4. For best performance, any user exit programs in the execution path used by the program must also be coded to threadsafe standards and defined to CICS as threadsafe. If any user exit programs in the execution path used by the program are not threadsafe, CICS switches to the QR TCB to run them, which might be detrimental to the performance of the application.

"Threadsafe programs" on page 232 explains the requirements for threadsafe programs in more detail.

OPENAPI programs have some additional obligations and restrictions. For example, they must ensure that all non-CICS resources acquired specifically on behalf of the terminating task are freed, and they must not use certain MVS system services. "Threadsafe restrictions for OPENAPI programs" on page 241 explains these requirements.

Candidate programs for defining as REQUIRED OPENAPI (assuming their application logic is threadsafe) are those programs that want to use other non-CICSAPIs at their own risk.

TCBs for OPENAPI programs

The following TCBs are used for OPENAPI programs:
- L8 mode TCBs are used for CICS key OPENAPI application programs, including some CICS programs that run on L8 TCBs for processing web services requests, parsing XML, and accessing z/OS UNIX files for CICS web support.
- L9 mode TCBs are used for user key OPENAPI application programs.

L8 mode TCBs are also used when programs need access to a resource manager through a TRUE enabled by using the OPENAPI option on the ENABLE PROGRAM command. An open API TRUE is given control under an L8 mode TCB, and can use non-CICS APIs without creating subtask TCBs.

CICS automatically controls the number of L8 and L9 TCBs in the pool of L8 and L9 mode open TCBs.

The use of OPENAPI programs can cause more TCB switching than ordinary threadsafe programs. If the OPENAPI program uses any EXEC CICS commands or user exit programs that are not threadsafe, causing a switch to the QR TCB, there is an extra switch, because CICS switches back to the open TCB to continue running the application logic. Additional TCB switching might be involved because of the requirement for the key of the TCB to be correct for OPENAPI programs. OPENAPI TRUEs always run in CICS key on an L8 TCB, so, for example, if a user key OPENAPI program runs on an L9 TCB but makes a DB2 call, CICS switches to an L8 TCB to call DB2, then returns to the L9 TCB to continue running the

program. Because of this switching, CICS DB2 applications are normally defined as (CICSAPI) threadsafe programs, rather than as OPENAPI programs. CICS key CICS DB2 applications can be defined as OPENAPI programs if required.

Threadsafe restrictions for OPENAPI programs

An OPENAPI program, although freed from the constraints imposed by the QR TCB, still has 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.

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

Running applications with programs defined as threadsafe to use OTE (for example, in CICS DB2 applications) could cause problems if one or more programs is not 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 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 PROGRAM definition attributes in the *CICS Resource Definition Guide*.

CICS 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.

For more information about CICS DSAs, refer to CICS dynamic storage areas in Improving performance.

Storing data in a transaction

CICS provides a variety of facilities to store data in and between transactions. Each one differs according to how available it leaves data to other programs in a transaction and to other transactions; in the way it is implemented; and in its overhead, recovery, and enqueuing characteristics.

Storage facilities that exist for the lifetime of a transaction include:

- Transaction work area (TWA)
- User storage (by GETMAIN commands issued without the SHARED option)
- COMMAREA
- Program storage

These areas are all main storage facilities whose source is the dynamic storage areas (DSAs), extended dynamic storage areas (EDSAs), or above-the-bar dynamic storage areas (GDSAs). None of them is recoverable, and none can be protected by resource security keys. They differ in accessibility and duration, so each storage facility 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 “Storage control” on page 579 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.

User storage is useful for large storage requirements that are variable in size or are shorter-lived than the transaction.

See "Storage control" on page 579 for information about how the USERDATAKEY and CICSDATAKEY options of the GETMAIN commands override the TASKDATAKEY option of the transaction resource definition.

The SHARED option of the GETMAIN commands 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 commands to ensure that your storage is in common storage.

Because of the processor overhead involved in GETMAIN commands, do not use user storage for a small amount of storage. Use the transaction work area (TWA) for the smaller amounts, or group them into a larger request. Although the storage acquired by GETMAIN commands can be held 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 265.

Information in COMMAREA is available only to the two participating programs, unless those programs explicity make the data available to other programs that might be invoked later in the transaction.

- When one program links to another, the COMMAREA can 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 might copy the specified COMMAREA into a new area of storage, because the invoking program and its control blocks might no longer be available after it transfers control.

In both cases, the address of the area is passed to the program that is receiving control, and the CICS command-level interface sets up addressability. See "Program control" on page 300 for further information.

When XCTL is used, CICS ensures that any COMMAREA is addressable by the program that receives it, by creating the COMMAREA in an area that conforms to the addressing mode of the receiving program. If the receiver is AMODE(24), the

COMMAREA is created below the 16 MB line, and if the receiver is AMODE(31), the COMMAREA is created above 16 MB but below 2 GB.

The COMMAREA is copied to USERKEY storage where necessary, depending on the addressing mode and EXECKEY attributes of the receiving program. See "Storage control" on page 579 for more information about 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" on page 295. 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 "Transferring data between programs using channels" on page 267.

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 queues

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 temporary storage queues. The items can be retrieved by the originating task, or by any other task, by using the symbolic name assigned to the temporary storage queue.

A temporary storage queue containing multiple items can be thought of as a small data set. Specific items (logical records) in a queue are referred to by relative position numbers. The items 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 scratchpad area.

Temporary storage queues are identified by symbolic names of up to 16 characters. To avoid conflicts caused by duplicate names, establish a naming convention. For example, the operator identifier or terminal identifier could be used as a suffix to each programmer-supplied symbolic name. The fact that temporary storage queues can be named as they are created provides a powerful form of direct access to saved data. You can access scratchpad areas for resources such as terminals and data set records by including the terminal name or record key in the queue name.

Compared with other methods to pass data from task to task, temporary storage queues can require more processor use. You use an EXEC CICS command for every interaction with temporary storage queues, and CICS must find or insert the data by using its internal index. The processor use with main temporary storage is therefore greater than with the CWA or TCTUA. With auxiliary storage, there is typically data set I/O as well. Shared temporary storage pools require temporary storage servers, and applications must access the coupling facility to retrieve the data.

However, temporary storage queues have a number of advantages over other methods to pass data. You do not need to allocate temporary storage until it is required. You keep it only as long as it is required, and the item size is not fixed until you issue the command that creates it. Temporary storage queues are therefore a good choice for relatively high-volume data and data that varies in length or duration. Resource protection is also available for temporary storage queues.

Temporary storage queues remain in storage until they are deleted by the originating task, by any other task, or on an initial or cold start. Your application can use the **DELETEQ TS** command to delete temporary storage queues at the end of their useful life. If your application cannot always delete temporary storage queues, consider setting up automatic deletion. You can make CICS automatically delete non-recoverable temporary storage queues in main storage or auxiliary storage if they have not been accessed recently. The expiry interval in your TSMODEL resource definitions controls automatic deletion.

Selecting a location for temporary storage queues

Temporary storage for a CICS region can be in main storage, auxiliary storage, or shared temporary storage pools in a z/OS coupling facility. For an overview of the different temporary storage locations and the features available for temporary storage queues in each location, see CICS temporary storage overview in Improving performance.

An application uses the **WRITEQ TS** command to write the first item of data to a temporary storage queue. The command specifies the symbolic name of the temporary storage queue. If this name matches an installed TSMODEL resource definition, CICS creates the temporary storage queue in the temporary storage location specified by the TSMODEL resource definition. If there is no matching TSMODEL resource definition, CICS uses the temporary storage location that the application specifies on the **WRITEQ TS** command. The default location is auxiliary temporary storage.

To choose where your temporary storage queues are located, consider these factors:

Lifetime and frequency of use

- Main temporary storage is more appropriate for temporary storage queues that are needed for short periods of time, or that are accessed frequently.
- Auxiliary temporary storage and shared temporary storage pools are more appropriate for temporary storage queues that have a relatively long lifetime, or are accessed infrequently. You could use a cutoff point of a 1 second lifetime to decide whether queues go in main storage, or auxiliary or coupling facility storage.

Recovery

- Temporary storage queues in main storage cannot be defined as recoverable.
- Temporary storage queues in auxiliary storage can be defined as recoverable.
- CICS recovery is not available for temporary storage queues in shared temporary storage pools. However, the coupling facility is not affected by a CICS restart, so temporary storage queues in shared temporary storage pools can be considered persistent.

Automatic deletion

You can specify that CICS deletes eligible temporary storage queues automatically when they are no longer required, by adding an expiry interval to the corresponding temporary storage models.

- You can set an expiry interval for temporary storage queues in main storage.
- You can set an expiry interval for nonrecoverable queues in auxiliary temporary storage. Recoverable queues cannot be deleted automatically.
- Temporary storage queues in shared temporary storage pools cannot be deleted automatically.

Storage type

Main temporary storage is in 64-bit storage in the CICS region. If you do not require recoverable temporary storage, you can specify that an application uses main temporary storage. As a result, there is less pressure on space in 31-bit storage, and reduced I/O activity to write data to disk.

Locking and waits for temporary storage queues

The CICS temporary storage domain can process multiple requests concurrently, but it serializes requests made for the same temporary storage queue. The queue is locked for the duration of each request.

Only one transaction at a time can write to or delete a recoverable temporary storage queue. If you choose to make queues recoverable, bear in mind the probability of enqueues.

If a task tries to write to temporary storage and there is no space available, CICS normally suspends it. The task can regain control 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 situation can produce unexplained response delays, especially if the waiting task owns exclusive-use resources, so all other tasks needing those resources must also wait.

Related reference

WRITEQ TS in Reference -> Application development

READQ TS in Reference -> Application development

DELETEQ TS in Reference -> Application development

TSMODEL resources in Reference -> System definition

Intrapartition transient data

Intrapartition transient data consists of queues of data, kept together in a single data set, with an index that CICS maintains in main storage.

Intrapartition transient data has some characteristics in common with auxiliary temporary storage. (See "Efficient sequential data set access" on page 260 for information about ***extrapartition*** transient data.)

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 or GETMAIN64 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 or FREEMAIN64 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 typically accepts the length as a signed halfword binary value. The use of a signed halfword binary value places a theoretical upper limit of 32 KB on the length. In practice, the limits are lower 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 Application development reference for programming information, including details of when you need to specify the LENGTH option. Where possible, do not let the length specified in CICS command options exceed 24 KB. For more information, see LENGTH options in CICS commands in Reference -> Application development.

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.

When you use an **EXEC CICS LINK** command to pass data in a COMMAREA, ensure that you specify a LENGTH value that matches the length of the data being passed in the COMMAREA. Do not specify 0 (zero) for LENGTH, because the resulting behavior is unpredictable and the **EXEC CICS LINK** command might fail. When you use a COMMAREA to pass data, the program that is linked to must verify that the EIBCALEN field in the EIB of the task matches what the program expects. Discrepancies might result in storage violations or system failures. For more information, see “COMMAREA” on page 302.

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 CICS-value data areas (cvdas) in Reference -> Application development.

For journal commands, the restrictions apply to the sum of the LENGTH and PFXLENG values. See “Journaling” on page 345.

Journal records

For journal records, the journal buffer size can impose a limit lower than 64 KB. 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 24 KB.

For information about creating data sets, see Defining data sets in Configuring. For information about resource definition for files, see FILE resources in Reference -> System definition.

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 to cause an error or to place a constraint on applications.

Note: The value in the LENGTH option must never exceed the length of the data area addressed by the command.

Minimizing errors

Use these techniques to help you make 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 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

Use these storage tools and techniques to minimize errors in your application programs.

About this task

- You can use the storage protection facility to prevent CICS code and control blocks from being overwritten by your application programs. You can choose whether to use this facility by using CICS system initialization parameters. For more information about this facility, see Storage protection in Improving performance.

- Consider using standards that avoid problems that can be caused by techniques such as the use of GETMAIN commands.

Testing applications

Some general rules that apply to testing applications are listed.

About this task

- Do not test on a production CICS system. Use a test system, where you can isolate errors without affecting live databases.
- Use someone other than the application developer to do the testing, if possible.
- Document the data you use for testing.
- Test your applications several times. See Chapter 11, “Testing applications,” on page 161.
- Use the CEDF transaction for initial testing. See Chapter 12, “Execution diagnostic facility (EDF),” on page 165.
- Use stress or volume testing to find problems that are not found in a single-user environment. A good tool for this is Teleprocessing Network Simulator (TPNS, licensed program number 5740-XT4).

  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 Multiregion operation.
- Before you move an application to the production system, 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 Approaches to problem determination in Troubleshooting.

Non-terminal transaction security

CICS can protect resources used in non-terminal transactions against unauthorized use.

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*.

Design for performance

You can change the design of your application program to improve performance and efficiency.

- "Program size" on page 253
- "Virtual storage" on page 254
- "Exclusive control of resources" on page 257
- "Operational control" on page 258
- "Operating system waits" on page 259
- "The NOSUSPEND option" on page 259
- "Efficient sequential data set access" on page 260
- "Efficient logging" on page 261

Other aspects of application design are addressed in Chapter 18, "Application design," on page 223

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

By careful design, you can minimize the amount of virtual storage used, and reduce your application's overhead.

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 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.

About this task

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*.

About this task

You can do this by:

- Making the execution of the program as linear as possible.
- Keeping any subroutines that 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 few 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 in 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. 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 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 previous suggestions can 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 can also result in more exceptions to sequential processing than are found in a nonstructured program. Nevertheless, the much greater productivity associated with structured code can 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.

About this task

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.

About this task

The reference set can be reduced 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

Do not attempt 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 like that for a real environment. The system should have all modules resident so that code on pages that are not referred to do not need to 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 fundamental and powerful recovery facilities that CICS provides have performance implications. You can adopt various approaches to reduce contention delays for resources.

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 unrecoverable 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 511 for information about the TOKEN keyword. Transient data, temporary storage, and user journals must be included among such resources. Locking resources in application programs in the *CICS Recovery and Restart Guide* contains further information about the extent of resource protection.

It might 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

You can use a number of operational techniques 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 Investigating loops that are not detected by CICS in 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 operating system waits are:

- Extrapartition transient data sets. (See "Efficient sequential data set access" on page 260.)
- 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 3, "Developing COBOL applications," on page 23, Chapter 4, "Programming in C and C++," on page 45, and Chapter 5, "Developing PL/I applications," on page 55.
- SVCs and assembler language macros that invoke operating system services, such as write-to-operator (WTO).

The NOSUSPEND option

The default action for certain commands is to suspend the application until the required resource becomes available. You can use the NOSUSPEND option to inhibit this wait and return to the next instruction in the application program.

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 of the command. The following commands can result in these conditions:

- ALLOCATE
- ENQ
- GETMAIN
- WRITE JOURNALNAME
- WRITE JOURNALNUM
- READQ TD
- WRITEQ

You can use the NOSUSPEND option (also known as the NOQUEUE option for the ALLOCATE command) to inhibit this wait and cause an immediate return to the instruction in the application program that follows the command.

CICS maintains a table of conditions referred to by the HANDLE CONDITION and IGNORE CONDITION commands in a COBOL application program.

Restriction: HANDLE CONDITION and IGNORE CONDITION commands are supported only in COBOL, PL/I, and assembler language applications (but not AMODE(64) assembler language applications). They are not supported in all other supported high level languages.

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 following three states:

- A label is currently specified:

```
HANDLE CONDITION condition(label)
```

- The condition is to be ignored:
  ```
  IGNORE CONDITION
  ```
- No label is currently specified:
  ```
  HANDLE CONDITION
  ```

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 entry 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 sequential data set access

CICS provides a number of different sequential processing options. Each has different performance characteristics.

Temporary storage and intrapartition transient data queues (already discussed in "Temporary storage queues" on page 245 and in "Intrapartition transient data" on page 248) 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 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 might 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.

Sharing data across transactions

CICS has several facilities for sharing data across transactions. You can use the Common Work Area (CWA), the TCTTE user area (TCTUA), the COMMAREA, the display screen, or 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 can write to them and any transaction can read them.

The use of some of these facilities might cause inter-transaction affinities. See "Affinity" on page 312 for more information about transaction affinities.

This section describes:

- "Using the common work area (CWA)"
- "Using the TCTTE user area (TCTUA)" on page 264
- "Using the COMMAREA in RETURN commands" on page 265
- "Using a channel on RETURN commands" on page 266
- "Using the display screen to share data" on page 266

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" 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 CWAKEY parameter *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).

Programs that run in CICS key can also write to CICS storage, ensure that such programs are thoroughly tested to make sure that they do not overwrite CICS storage.

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 "Storage control" on page 579.

Figure 64 on page 263 illustrates a particular use of the CWA where the CWA itself is protected from user-key application programs by CWAKEY=CICS.

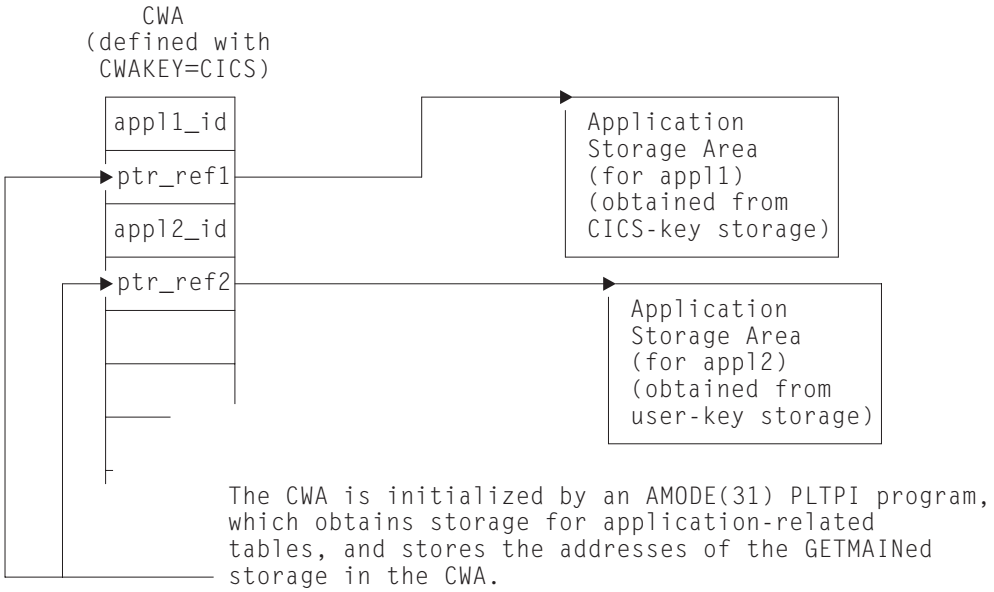


Figure 64. Example of use of CWA in CICS-key storage. This illustrates how the CWA can be used to reference storage that is obtained in user-key or CICS-key storage for use by application programs, while the CWA itself is protected by being in CICS-key storage.

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), ensuring the CWA is protected from overwriting by application programs. In Figure 64, 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 65 on page 264, 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.
* 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 65. Sample code for loading the CWA

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 Autoinstalling model terminal definitions in 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 244). 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, z/OS Communications Server for SNA 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 Communications Server. 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 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" on page 295. To pass a channel on a RETURN command, you use the CHANNEL option in place of the COMMAREA option.

Channels are described in "Transferring data between programs using channels" on page 267.

Using the display screen to share data

Data can be stored between pseudoconversational transactions from a 3270 display terminal on the display screen itself.

For example, errors made by users in data entry are highlighted (with highlights or messages) by the transaction processing the data. The next transaction identifier is then set to point to itself (so that it processes the corrected entry), 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 easy to code, but it is not secure. You are ***not*** recommended to save screens that contain large amounts of data as errors can 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. This can be avoided by defining the CLEAR key to mean CANCEL or QUIT, if appropriate for the application concerned.

Data other than keyed data can also be stored on the screen. This data can be protected from changes (except those caused by CLEAR) and can be nondisplay, if necessary.

Transferring data between programs using channels

Channels are a method of transferring data between programs, in amounts that far exceed the 32 KB limit that applies to COMMAREAs.

This section contains:

- "Channels: quick start"
- "Using channels: some typical scenarios" on page 270
- "Creating a channel" on page 273
- "The current channel" on page 274
- "The scope of a channel" on page 279
- "Discovering which containers were passed to a program" on page 282
- "Discovering which containers were returned from a link" on page 283
- "CICS read-only containers" on page 283
- "Designing a channel: Best practices" on page 284
- "Constructing and using a channel: an example" on page 285
- "Channels and BTS activities" on page 286
- "Using channels from JCICS" on page 288
- "Dynamic routing with channels" on page 288
- "Data conversion" on page 289
- "Benefits of channels" on page 295
- "Migrating from COMMAREAs to channels" on page 296

Channels: quick start

A brief introduction to channels and containers.

Channels and containers

Containers are named blocks of data designed for passing information between programs. 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 called 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.

AMODE (64) programs can use channels and containers to transfer data in 64-bit storage in the same way, by using **EXEC CICS PUT64 CONTAINER** and **EXEC CICS GET64 CONTAINER** commands. These commands are for use only in non-Language Environment (LE) AMODE(64) assembler language application programs, and CICS business transaction services (BTS) containers are not supported.

Channel containers are not recoverable. Pseudoconversational transactions started using RETURN TRANSID CHANNEL() cannot be restarted. If you have to use recoverable containers, use CICS business transaction services (BTS) containers.

Basic examples

A simple example of a program that creates a channel and passes it to second program.

Figure 66 on page 269 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 channels and containers are created 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 66. A simple example of a program that creates a channel and passes it to a second program

Figure 67 on page 270 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 67. 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 66 on page 269.

Using channels: some typical scenarios

Channels and containers provide a powerful way to pass data between programs. These scenarios show some examples of how channels can be used.

One channel, one program

This example shows a stand-alone program with a single channel.

Figure 68 on page 271 shows the simplest scenario—a “stand-alone” program with a single channel with which it can be invoked.

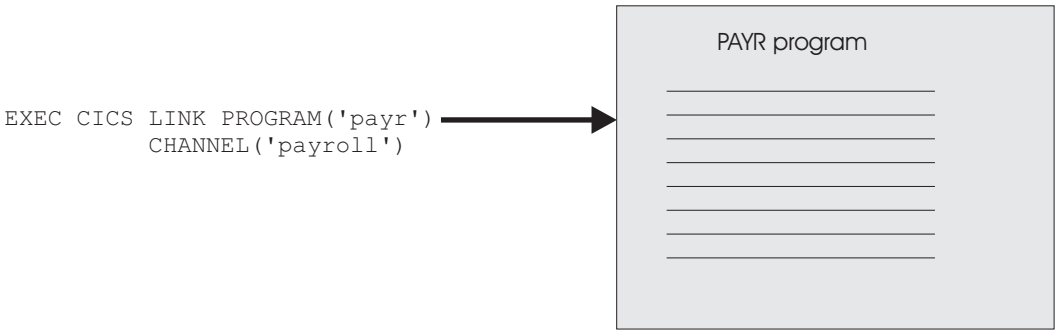


Figure 68. A stand-alone program with a single channel

One channel, several programs (a component)

This example shows a set of related programs (a component) invoked through a single channel.

In Figure 69, 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.

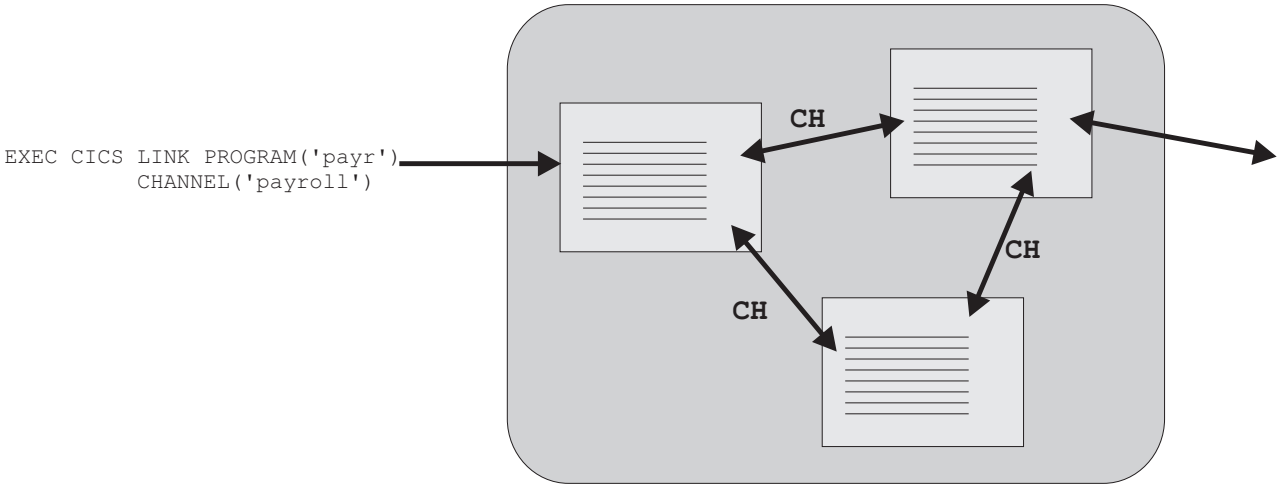


Figure 69. A "component"—a set of related programs invoked through a single external channel. "CH" indicates that the programs within the component can pass channels between themselves.

Several channels, one component

This example shows a set of related programs (a component) which can be invoked through two alternative channels.

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.

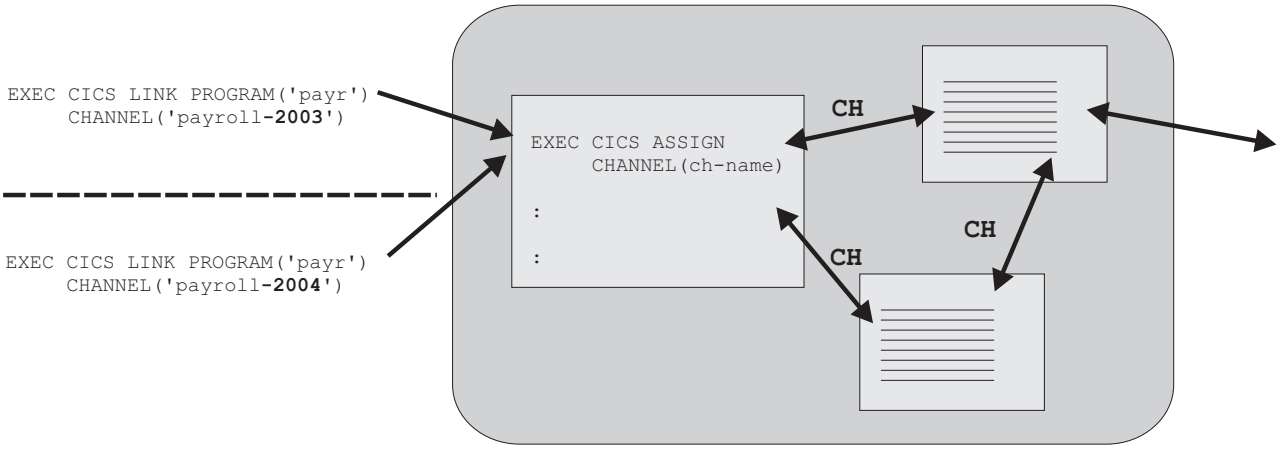


Figure 70. Multiple external channels to the same component. "CH" indicates that the programs within the component can pass channels between themselves.

Multiple interactive components

This example shows how multiple components can interact through their channels.

Figure 71 on page 273 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 stand-alone program and the Human resources component.

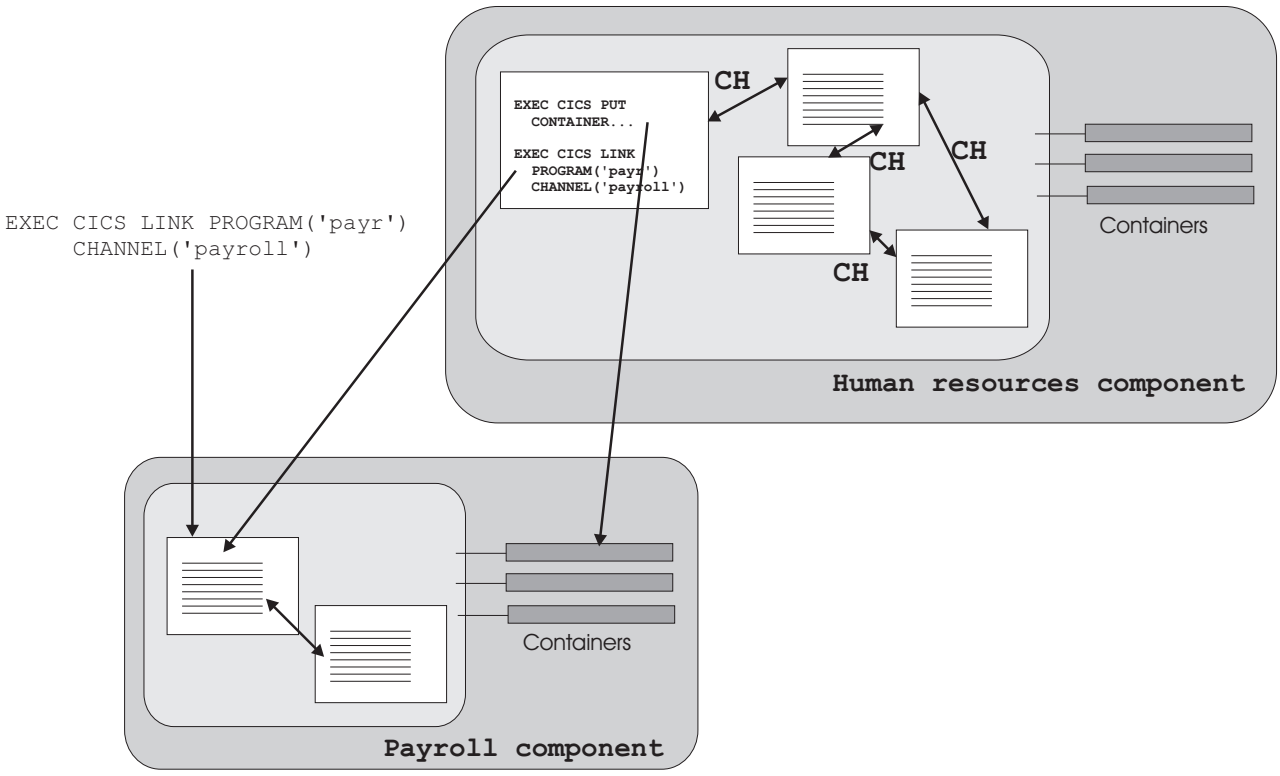


Figure 71. Multiple components which interact through their channels

Creating a channel

You can create a channel by naming it on one of a number of API commands. If the channel does not exist in the current program scope, CICS creates it.

About this task

You create a channel by naming it on one of the following commands:

LINK PROGRAM CHANNEL
MOVE CONTAINER CHANNEL TOCHANNEL
PUT CONTAINER CHANNEL
PUT64 CONTAINER
RETURN TRANSID CHANNEL
START TRANSID CHANNEL
XCTL PROGRAM CHANNEL

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 does not exist), and adds a container to it; the subsequent commands add further containers to the channel. If the containers exist, their contents are overwritten by the new data.

AMODE (64) programs can create and use channels in the same way, by using **EXEC CICS PUT64 CONTAINER** commands. This command is for use only in non-Language Environment (LE) AMODE(64) assembler language application programs.

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 is not specified, the current channel is implied.
2. The source channel must be in program scope.
3. If the target channel does not exist in the current program scope, it is created.
4. If the source container does not exist, an error occurs.
5. If the target container does not exist, it is created; if the target container exists, its contents are overwritten.
6. When a channel is created, it exists until the task that created it terminates. For example, if a long running task performs many PUT CONTAINER commands to different unique channels, all the channels that are created do not release the main storage acquired by the GETMAIN command until the task ends. You must design your applications accordingly to prevent your regions becoming short on storage.

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 does not exist in 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 there is one - with which it was invoked. These examples show how the current channel and its containers are passed between programs.

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

This example shows how a program passes the current channel and its containers to another program using the EXEC CICS LINK command.

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 is not started by a program and does not have a current channel.

B, C, D, and E are second-, third-, fourth-, and fifth-level programs.

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 does not pass it one.

Program E's current channel is Y, passed by the CHANNEL option on the EXEC CICS LINK command issued by D.

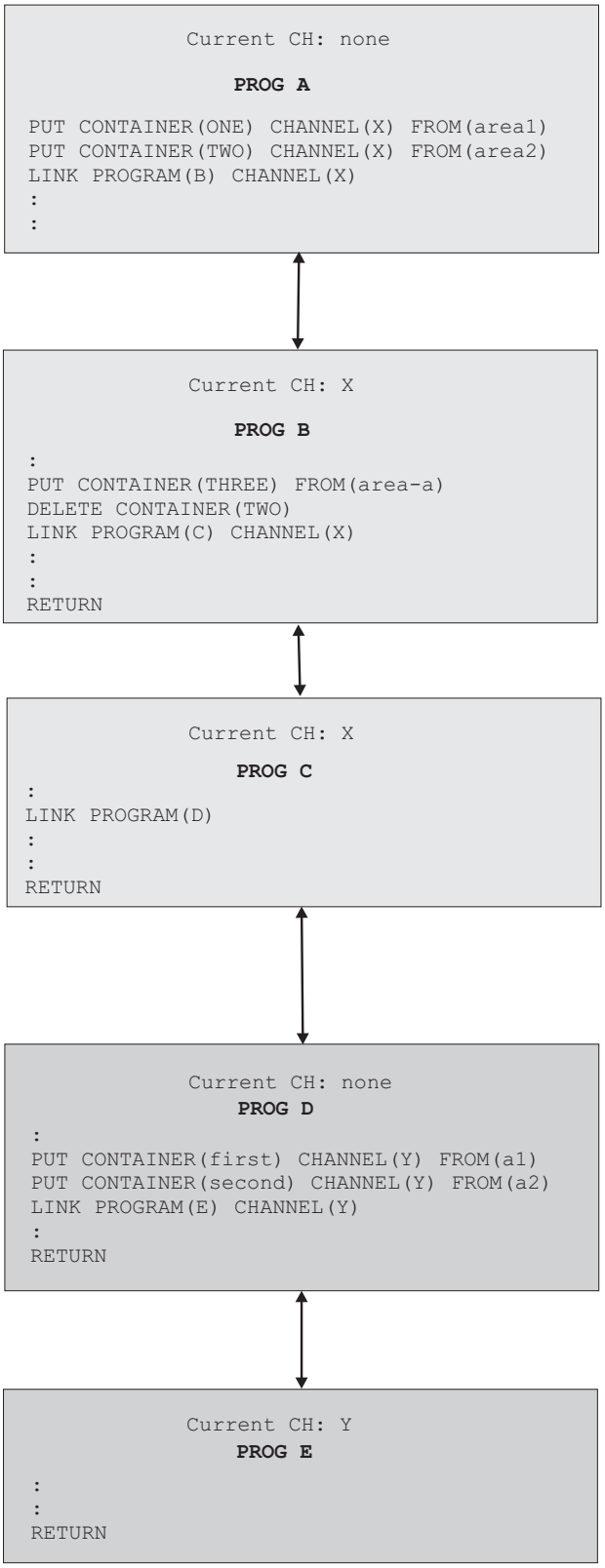


Figure 72. Current channel: example with LINK commands

The following table lists the name of the current channel (if any) of each of the five programs shown in the previous figure.

Table 17. The current channels of interactive programs—example with LINK commands

Prog.	Current CH	Issues commands	Comments

A	None	`.`
		`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)`
		`.`
			Program A creates channel X and passes it to program B.
			
			Note that, by the time control is returned to program A by program B, the X channel has been modified—it does not 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.)

B	X	`.`
		`EXEC CICS PUT CONTAINER(THREE)`
		`FROM(area-a)`
		`EXEC CICS DELETE CONTAINER(TWO)`
		`EXEC CICS LINK PROGRAM(C) CHANNEL(X)`
		`.`
		`.`
		`EXEC CICS RETURN`
			Program B modifies channel X (its current channel) by adding and deleting containers, and passes the modified channel to program C.
			
			Program B does not need to specify the CHANNEL option on the PUT CONTAINER and DELETE CONTAINER commands; its current channel is implied.

C	X	`.`
		`EXEC CICS LINK PROGRAM(D)`
		`.`
		`.`
		`EXEC CICS RETURN`
			Program C links to program D, but does not pass it a channel.

D	None	`.`
		`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`
			Program D creates a new channel, Y, which it passes to program E.

E	Y	`.`
		`RETURN`
		`.`
			Program E performs some processing on the data it's been passed and returns.

Current channel example, with XCTL commands

This example shows how a program passes the current channel and its containers to another program using the EXEC CICS XCTL command.

Figure 73 on page 278 shows four interactive programs. A1 is a top-level program started by, for example, a terminal end user. It is not started by a program and does not 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 does not 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.

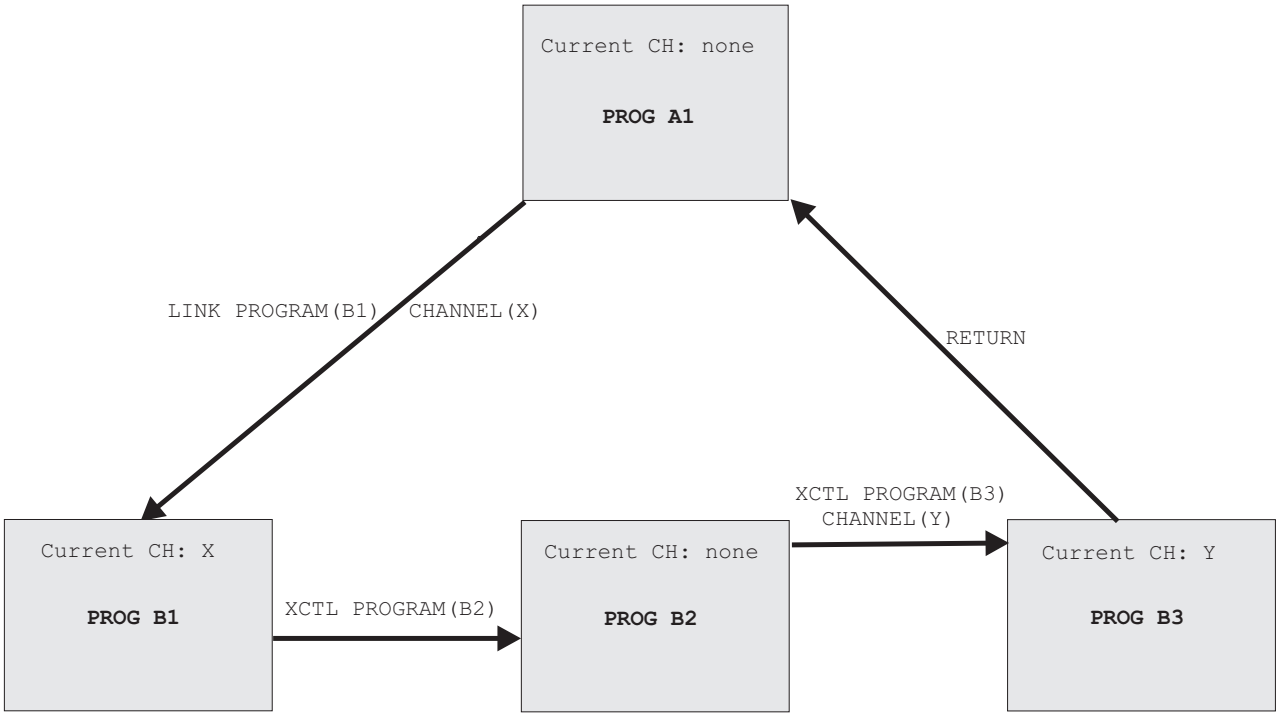


Figure 73. Current channels—example, with XCTL commands

The following table lists the name of the current channel (if any) of each of the four programs shown in Figure 73.

Table 18. 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

As well as the **LINK** and **XCTL** commands, you can pass channels on the **START** and **RETURN** commands.

EXEC CICS START TRANSID(*tranid***) 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. These examples show the scope of each channel in the diagram.

Scope example, with LINK commands

This example adds to the diagram from "Current channel example, with LINK commands" to show the scope of each channel.

The following figure shows the same five interactive programs previously described.

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 does not contain the same set of containers as when it was created by program A.

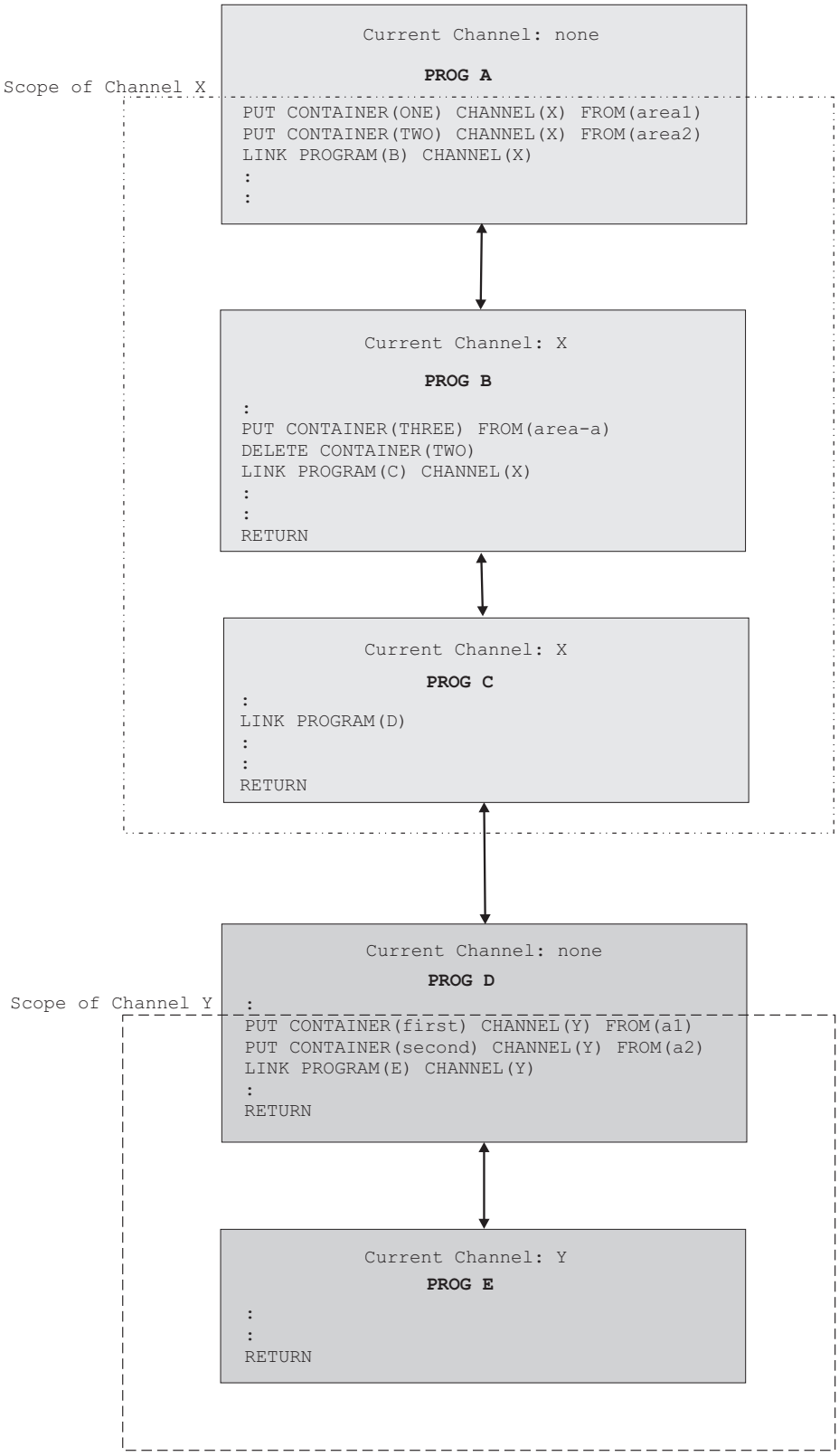


Figure 74. The scope of a channel—example showing LINK commands

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 19. 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

This example adds to the diagram from "Current channel example, with XCTL commands" to show the scope of each channel.

Figure 75 on page 282 shows the same four interactive programs previously described, 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 is possible that the X channel can have been modified by program B1, it might not contain the same set of containers as when it was created by A1.

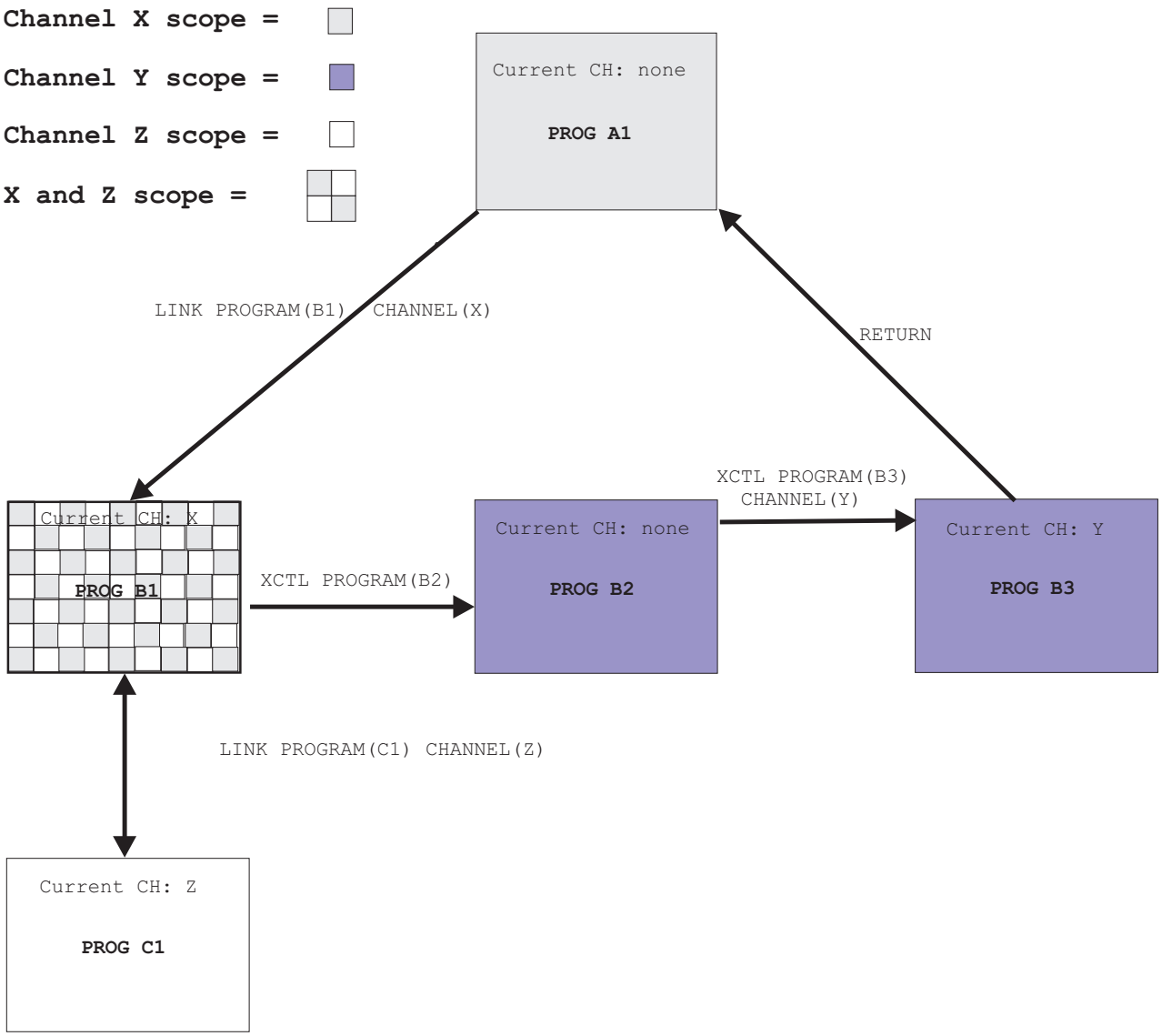


Figure 75. The scope of a channel—example showing LINK and XCTL commands

The following table lists the name and scope of the current channel (if any) of each of the five programs shown in Figure 75.

Table 20. 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 were passed to a program

When a program is invoked, it can determine the names of the channel, and any containers that were passed to it.

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 were passed to it.

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 was passed to it.

Discovering which containers were returned from a link

Following a LINK command, a program can discover the names of the containers returned by the program that was linked to.

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 that was passed to it, 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 that 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**, **PUT64 CONTAINER**, **MOVE CONTAINER**, or **DELETE CONTAINER** command, an INVREQ condition occurs.

Designing a channel: Best practices

Containers are used to pass information between programs. These containers are grouped together in sets called channels. It is advisable to design your channels to follow a set of best practices.

Procedure

- At the end of a DPL call, input containers that the server program has not changed are not returned to the client. Input containers whose contents have been changed by the server program, and containers that the server program has created, are returned. Therefore, for optimal DPL performance, use the following best practices:
  - Use separate containers for input and output data.
  - Ensure that the server program, not the client, creates 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.
- The following general tips on designing a channel include, and expand on, the recommendations to achieve optimal DPL performance:
  - Use separate containers for input and output data. This provides the following benefits:
    - 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.
  - Ensure that the server program, not the client, creates the output containers. If the client creates them, empty containers are sent to the server region.
  - Use separate containers for read-only and read-write data. This provides the following benefits:
    - 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 provides the following benefits:
    - Better encapsulation of the data, making your programs easier to understand and maintain.
    - Greater ease in changing one of the structures, because you do not need to recompile the entire component.
    - The ability to pass a subset of the channel to subcomponents, 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 provides the following benefits:
    - Easier identification of error information.
    - Greater efficiency, for the following reasons:
      - The structure containing the error information is passed back only if an error occurs.
      - It is more efficient to check for the presence of an error container by issuing a **GET CONTAINER**(*known-error-container-name*) or **GET64**

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 might affect performance. This is particularly true if the local and remote regions are connected by an ISC, rather than MRO, connection.

  Attention: Take care not to create so many large containers that you limit the amount of storage available to other applications.
- Channels and containers use storage below 2 GB (below the bar) and some 64-bit (above-the-bar) storage. Their use of 64-bit storage influences the value that you choose for the z/OS **MEMLIMIT** parameter that applies to the CICS region. You must also allow for other CICS facilities that use 64-bit storage. For more information, see Estimating, checking, and setting MEMLIMIT in Improving performance.

Related concepts:

"Migrating from COMMAREAs to channels" on page 296
CICS application programs that use traditional communications areas (COMMAREAs) to exchange data can continue to work as before. If you want to migrate to channels, here are examples of how to migrate several types of existing application to use channels and containers rather than COMMAREAs.

Related reference:

"Benefits of channels" on page 295
Using the channel and container model in CICS programs to exchange data has several advantages over using communication areas (COMMAREAs) .

Constructing and using a channel: an example

In this example, a client program constructs a channel, passes it to a server program, and retrieves the server's output. The server program retrieves data from the channel's containers, and returns output to the client.

Figure 76 on page 286 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 76. How a client program can construct a channel, pass it to a server program, and retrieve the server's output

Figure 77 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 77. 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.

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 78.

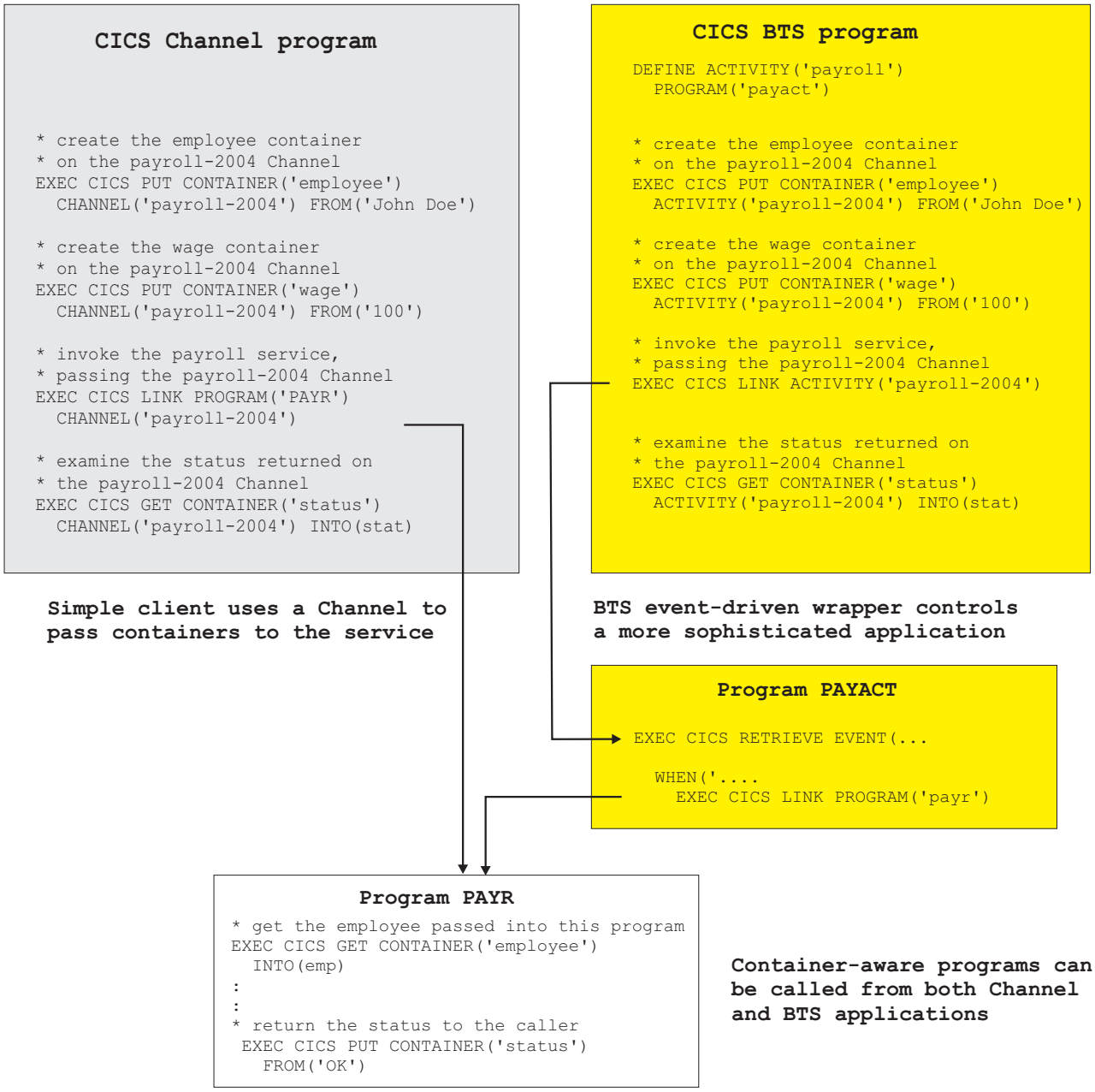


Figure 78. Channels and BTS activities

Context

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

CICS provides JCICS classes that CICS Java programs can use to pass and receive channels

For information about using channels with JCICS, see Channels and containers in *Java Applications in CICS*.

Dynamic routing with channels

EXEC CICS LINK and EXEC CICS START commands, which pass 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 as previously listed. 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.

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.

Related information:

Introduction to CICS dynamic routing

Writing a dynamic routing program

Data conversion

Application programs that use channels and containers frequently need to convert data from one code page to another.

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. For an explanation of CCSIDs, and a list of the CCSIDs supported by CICS, see CICS-supported conversions in Reference -> Connectivity and standards.

Preparing for code page conversion with channels

CICS supports character conversions with the z/OS conversion services.

Before you begin

If you are preparing for code page conversions that involve Unicode data, see the information in Enabling Unicode data conversion by z/OS in the *CICS Transaction Server for z/OS Installation Guide*.

About this task

The conversion of data to or from either UTF-8 or UTF-16 and EBCDIC and ASCII code pages, depends on the selection of suitable conversion images. Conversion between the UTF-8 and UTF-16 forms of Unicode is also supported.

See the relevant appendix in z/OS Unicode Services User's Guide and Reference for the conversions that are supported though these services. These conversions 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 WebSphere MQ transport, is an EBCDIC to EBCDIC conversion resulting from small differences in the character encodings used by CICS and WebSphere MQ.
2. Not all points in each code page have direct counterparts in other code pages, for example the EBCDIC character NL. Java and z/OS conversion services might differ in the conversions that they perform. Technotes and other Internet discussions might offer guidance on particular points. Programming communities themselves are divided on the question of what is the more appropriate conversion in particular circumstances.

CICS now supports any of these character conversions by using the z/OS conversion services. Earlier releases of CICS used a set of tables to carry out conversions, and conversions that use those tables continue to be supported. If CICS needs to carry out a conversion between a pair of CCSIDs that are not supported by these tables, the z/OS conversion services are used.

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 CCSIDs: 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 recognized 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 use the 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

Code page 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. For an example, see Putting data into a container in *Java Applications in CICS*.

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 simple. The data in channel containers is converted under the control of the application programmer, by using API commands.
- The application programmer is responsible only for the conversion of user data; that is, the data in containers created by the 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.

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 was 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. For more information about DBCS, see the IBM Glossary.

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.

There are two ways to specify the code page for data conversion of the data in a container:

- As a Coded Character Set Identifier, or CCSID. A CCSID is a decimal number that identifies a specific code page. For example, the CCSID for the ASCII character set ISO 8859-1 is 819.
- As an IANA-registered charset name for the code page. This is an alphanumeric name that can be specified in charset= values in HTTP headers. For example, the IANA charset names supported by CICS for ISO 8859-1 are `iso-8859-1` and `iso_8859-1`.

If the application programmer does not specify a code page for data conversion, CICS uses the default code page for the whole of the local CICS region, which is specified on the **LOCALCCSID** system initialization parameter.

You can use the following API commands for data conversion:

- ```
  EXEC CICS PUT CONTAINER [CHANNEL] [DATATYPE] [FROMCCSID | FROMCODEPAGE]
  ```
- ```
  EXEC CICS GET CONTAINER [CHANNEL] [INTOCCSID | INTOCODEPAGE]
  ```

For non-Language Environment (LE) AMODE(64) assembler language application programs, you can also use the following API commands for data conversion for data in 64-bit storage:

- ```
  EXEC CICS PUT64 CONTAINER [CHANNEL] [DATATYPE] [FROMCCSID | FROMCODEPAGE]
  ```
- ```
  EXEC CICS GET64 CONTAINER [CHANNEL] [INTOCCSID | INTOCODEPAGE]
  ```

How to cause CICS to convert character data automatically:

You can use the DATATYPE(DFHVALUE(CHAR)) option of the PUT CONTAINER or PUT64 CONTAINER command to specify that a container holds character data eligible for conversion. If the client and server platforms are different, the GET CONTAINER or GET64 CONTAINER command converts the data automatically.

About this task

The following procedure demonstrates how to convert character data automatically by using PUT CONTAINER and GET CONTAINER commands.

For AMODE(64) programs, you can convert character data that is in 64-bit storage in the same way by using PUT64 CONTAINER and GET64 CONTAINER commands. These commands are for use only in non-Language Environment (LE) AMODE(64) assembler language application programs, and CICS business transaction services (BTS) containers are not supported.

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 or FROMCODEPAGE 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 or INTOCODEPAGE 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 or FROMCODEPAGE 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 or INTOCODEPAGE 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.

Using containers to do code page conversion:

Your applications can use the container API as a simple means of converting character data from one code page to another.

About this task

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.)
- If you prefer to specify a supported IANA charset name for the code pages, rather than the decimal CCSIDs, or if you want to specify a CCSID alphanumerically, use the FROMCODEPAGE and INTOCODEPAGE options instead of the FROMCCSID and INTOCCSID options.
- To avoid a storage overhead, after conversion copy the converted data and delete the container.
- To avoid shipping the channel, use a temporary channel. For more information about temporary storage and use of channels, see "Avoiding affinities when using temporary storage" on page 327.
- 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:

You can use a CICS TS SOAP application to retrieve a UTF-8 or UTF-16 message from a socket or WebSphere MQ message queue, put a message into a container in UTF-8 format, put EBCDIC data structures into other containers on the same channel, or make 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 UTF-8 or UTF-16, 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 UTF-8, UTF-16, 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 and INTOCODEPAGE options are 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 is permitted. Because the FROMCCSID and FROMCODEPAGE options are not specified, the message data is taken to be in the region's EBCDIC code page.

To retrieve one of the messages in UTF-8, the INTOCCSID or INTOCODEPAGE option must be specified to identify the code page and prevent automatic conversion of the data to the region's EBCDIC code page. The handler can issue the command:

```
EXEC CICS GET CONTAINER(input_msg) INTO(msg) INTOCCSID(utf8)
```

In this case, `utf8` is a variable which is defined as a fullword, and is initialized to 1208, which is the Coded Character Set Identifier, or CCSID, for UTF-8. If you prefer to use an IANA charset name for the code page, you can use the INTOCODEPAGE option instead of the INTOCCSID option:

```
EXEC CICS GET CONTAINER(input_msg) INTO(msg) INTOCODEPAGE(utf8)
```

In this case, `utf8` is a variable which is defined as a character string of length 56, and is initialized to 'utf-8'.

To return some output in UTF-8, the server program can issue the command:

```
EXEC CICS PUT CONTAINER(output) FROM(output_msg) FROMCCSID(utf8)
```

or alternatively:

```
EXEC CICS PUT CONTAINER(output) FROM(output_msg) FROMCODEPAGE(utf8)
```

where the variable `utf8` is defined and initialized in the same ways as for INTOCCSID and INTOCODEPAGE. The FROMCCSID or FROMCODEPAGE option specifies that the message data is currently in UTF-8 format. Because FROMCCSID or FROMCODEPAGE is specified, a DATATYPE of CHAR is implied, so data conversion is permitted.

Benefits of channels

Using the channel and container model in CICS programs to exchange data has several advantages over using communication areas (COMMAREAs) .

- 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.
- A channel can comprise multiple containers, passing data in a structured way.
- Channels do not 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 might be remote.
- Channels can be used by CICS application programs written in any of the language supported by CICS. 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 has 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 simple and is controlled by the application developer, using simple API commands.
- Programmers with experience of CICS business transaction services (BTS) can 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.)

Channels might not be the best solution in all cases. When designing an application, there are one or two implications of using channels that you must consider:

- When a channel is to be passed to a remote program or transaction, passing a large amount of data might affect performance, particularly if the local and remote regions are connected by an ISC, rather than MRO, connection.
- A channel might use more storage than a COMMAREA designed to pass the same data because container data can be held in more than one place. COMMAREAs are accessed by pointers, whereas the data in containers is copied between programs.

Migrating from COMMAREAs to channels

CICS application programs that use traditional communications areas (COMMAREAs) to exchange data can continue to work as before. If you want to migrate to channels, here are examples of how to migrate several types of existing application to use channels and containers rather than COMMAREAs.

It is 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 is not good practice to do this. 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 284. Channels have several advantages over COMMAREAs (see “Benefits of channels” on page 295) and it pays to design your channels to make the most of these improvements.

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 295.)

User-written dynamic or distributed routing programs require work whether or not you plan to implement channels and containers in your own applications. 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 DYRLEVEL, DYRTYPE, and DYRVER fields of the DFHDYPDS communications area. See Parameters passed to the dynamic routing program in the *CICS Customization Guide*.

Migrating LINK commands that pass COMMAREAs

To migrate two programs which use a COMMAREA on a LINK command to exchange a structure, change the instructions shown in this table.

About this task

In these instructions, `structure` is the name of your defined data structure. The EXEC CICS GET CONTAINER and PUT CONTAINER commands use a 16–character field to identify the container. A helpful convention is to give the container the same name as the data structure that you are using, shown here as `structure-name`.

Table 21. 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

To migrate two programs which use a COMMAREA on an XCTL command to pass a structure, change the instructions shown in this table.

About this task

In these instructions, `structure` is the name of your defined data structure. The EXEC CICS GET CONTAINER and PUT CONTAINER commands use a 16–character field to identify the container. A helpful convention is to give the container the same name as the data structure that you are using, shown here as `structure-name`.

Table 22. 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)`
		⋮

Table 22. Migrating XCTL commands that pass COMMAREAs (continued)

Program	Before	After

PROG2	`EXEC CICS ADDRESS COMMAREA(structure-ptr)`
	`...`
		`EXEC CICS GET CONTAINER(structure-name)`
		`INTO(structure)`
		`...`

Migrating pseudoconversational COMMAREAs on RETURN commands

To migrate two programs which use COMMAREAs to exchange a structure as part of a pseudoconversation, change the instructions shown in this table.

About this task

In these instructions, `structure` is the name of your defined data structure. The EXEC CICS GET CONTAINER and PUT CONTAINER commands use a 16–character field to identify the container. A helpful convention is to give the container the same name as the data structure that you are using, shown here as `structure-name`.

Table 23. 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

To migrate two programs which use START data to exchange a structure, change the instructions shown in this table.

About this task

In these instructions, `structure` is the name of your defined data structure. The EXEC CICS GET CONTAINER and PUT CONTAINER commands use a 16–character field to identify the container. A helpful convention is to give the container the same name as the data structure that you are using, shown here as `structure-name`.

Table 24. 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

In older CICS releases where channels were not available, some applications used temporary storage queues (TS queues) to pass more than 32K of data from one program to another. Typically, this involved multiple writes to and reads from a TS queue.

About this task

If you migrate one of these applications to use channels, be aware of the following points:

- 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. You can migrate these commands to use a channel in place of a COMMAREA.

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.

For introductory information about dynamic and distributed routing, see Introduction to CICS dynamic routing in the *CICS Intercommunication Guide*. For

information about writing a dynamic or distributed routing program, see Writing a dynamic routing program in the *CICS Customization Guide*.

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 here is the 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 programming using JCICS in Developing applications and the JCICS Javadoc documentation. For information about C++ programs using the CICS C++ classes, see Using the CICS C++ OO foundation classes in Developing applications.

Program control using the CICS API

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 as follows:

- You can 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.
- You can 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 "CICS intercommunication" on page 369 for more information about DPL.
- You can 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 "CICS intercommunication" on page 369 for more information about DPL.
- You can 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 "CICS intercommunication" on page 369 for more information about DPL.
- You can 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 more information about this utility and see "Affinity" on page 312 for more information about transaction affinity.

- You can 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.

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 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 regarding 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. Figure 79 on page 302 shows the concept of logical levels.

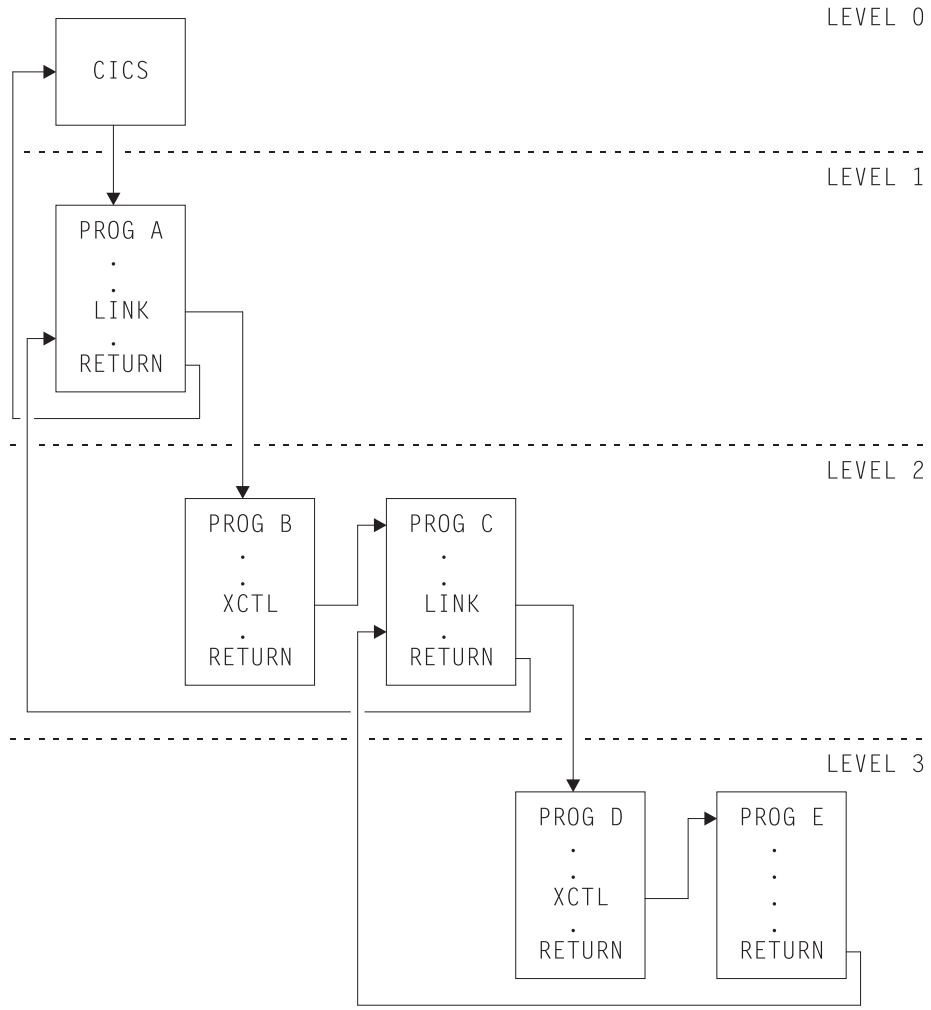


Figure 79. Application program logical levels

Passing data to other programs

You can pass data to another program by using the EXEC CICS program control commands LINK, XCTL, and RETURN, and using the COMMAREA, CHANNEL, or INPUTMSG options of those commands. COMMAREA and CHANNEL are mutually exclusive.

COMMAREA

The COMMAREA specifies the name of a data area (known as a *communication area*) in which data is passed to a program or transaction. It is an option of the LINK, XCTL, and RETURN commands.

The COMMAREA option of LINK and XCTL commands specifies the name of a data area in which data is passed to the program being invoked.

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 mapping. The register used to address this data area must be loaded from DFHEICAP (communication area pointer), mapped by the DFHEISTG DSECT. A COMMAREA cannot be in 64-bit storage.

The receiving data area can the same length as, or shorter than, the original communication area, but must not be longer. If access is required only to the first part of the data, the receiving data area can be shorter. If the receiving data area is longer than the original communication area, your transaction might attempt to read data outside the area that has been passed. It might also overwrite data outside the area, which might 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. Ensure that the value in EIBCALEN matches the value in the DSECT for your program, and ensure that your transaction accesses data in that area.

You can 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 by 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 by 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 that 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 can 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” on page 295. Channels can be passed, instead of COMMAREAs, on LINK, XCTL, and RETURN commands.

Channels are described in “Transferring data between programs using channels” on page 267.

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 with a LINKcommand 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 80 for an illustration of INPUTMSG.

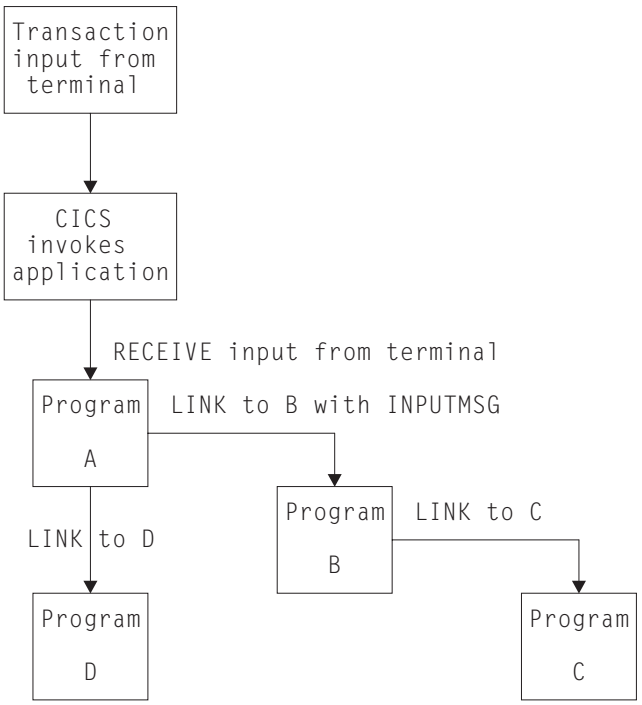


Figure 80. Use of INPUTMSG in a linked chain

Note:

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 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 LINKchain 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 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.

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 Writing a dynamic routing program in 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 within a communication area to an AMODE(64) program can be 64 bits or 31 bits long. You must change any 31-bit addresses to 64-bit addresses before use.
- Addresses passed as data to an AMODE(24) program must be below the 16 MB line if they are to be accessed 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 16 MB but below 2 GB (above the line) can be passed to an AMODE(24) subprogram. CICS copies the communication area into an area below 16 MB for processing. It copies it back again when control returns to the linking program.

CICS does not validate any data addresses passed within a communication area between programs with different addressing modes.

A COMMAREA cannot be in 64-bit storage.

Using LINK to pass data

These examples demonstrate how a LINK command causes data to be passed to the program being linked to. An XCTL command is coded in a similar way.

About this task

Figure 81 to Figure 84 on page 309 show how a LINK command is used in COBOL, C, C++, PL/I, and assembler language.

These examples show data being passed in a COMMAREA. For an example of a LINK command that uses a channel to pass data, see "Transferring data between programs using channels" on page 267.

Ensure that the LENGTH value you specify in the LINK command matches the length of the data being passed in the COMMAREA. Do not specify 0 (zero) for LENGTH, because the resulting behavior is unpredictable and the EXEC CICS LINK command might fail.

When you use a COMMAREA to pass data, the program that is linked to must verify that the EIBCALEN field in the EIB of the task matches what the program expects. Discrepancies might result in storage violations or system failures. For more information, see "COMMAREA" on page 302.

```
                    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 81. LINK command: COBOL example

In the following example, the COMMAREA contains a character string. For an example of a COMMAREA that contains a structure, see "Using RETURN to pass data" on page 309.

```
                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 82. LINK command: C example

```
                   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 83. LINK command: PL/I example

```
                  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 84. LINK command: assembler language example

Using RETURN to pass data

These examples demonstrate how a RETURN command is used to pass data to a new transaction.

About this task

Figures Figure 85 on page 310 to Figure 88 on page 312 show how a RETURN command is used in COBOL, C, C++, PL/I, and assembler language.

These examples show data being returned in a COMMAREA. For an example of a RETURN command that uses a channel to return data, see "Transferring data between programs using channels" on page 267.

```
                  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 85. RETURN command: COBOL example

```
                    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 86. RETURN command: C example

```
                   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 87. RETURN command: PL/I example

```
                  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 88. RETURN command: assembler language example

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 and is said to create an affinity between 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 executing in a CICSplex. For guidance on developing Java applications, see Java programming using JCICS in Developing applications.

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.
- Eligible non-terminal-related EXEC CICS START requests.

For detailed introductory information about dynamic and distributed routing, see Introduction to CICS dynamic routing in the *CICS Intercommunication Guide*.

Important:

The following sections talk exclusively about affinities between transactions.

- 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.

Types of affinity

Two types of affinity affect dynamic routing; inter-transaction affinity and transaction-system affinity. Inter-transaction affinity occurs among a set of transactions that share a common resource or coordinate their processing. Transaction-system affinity is an affinity between a transaction and a particular CICS region, where the transaction interrogates or changes the properties of that CICS region.

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 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 typically 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 that 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 ENQMODEL resource definitions *CICS Resource Definition Guide*.

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 with 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

In terms of inter-transaction affinity, the programming techniques used by your application programs in a dynamic or distributed routing environment, can be classed as safe, unsafe, or suspect.

- Safe techniques do not cause inter-transaction affinities.
- Unsafe techniques cause inter-transaction affinities inherently.
- Suspect techniques might, or might not create affinities depending on exactly how they are implemented.

Safe techniques

Safe programming techniques avoid the need for affinities. The programming techniques in the 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 regarding 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 319 and ENQMODEL resource definitions in 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 "Programming techniques that avoid affinity" on page 316.

Unsafe techniques

Unsafe programming techniques are techniques that can require the creation of affinities. 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 324 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 319 and the *CICS Resource Definition Guide* for a description of ENQMODELs.

For more information about unsafe programming techniques, see "Programming techniques that create affinities" on page 320.

Suspect techniques

Some programming techniques might, or might 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 “Programming techniques that might create affinities” on page 327.

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 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.

Programming techniques that 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.

To avoid creating affinities, do not use COMMAREAs, TCTUAs, and BTS containers to 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 89 on page 317.

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 24 KB (1 KB = 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 can 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.

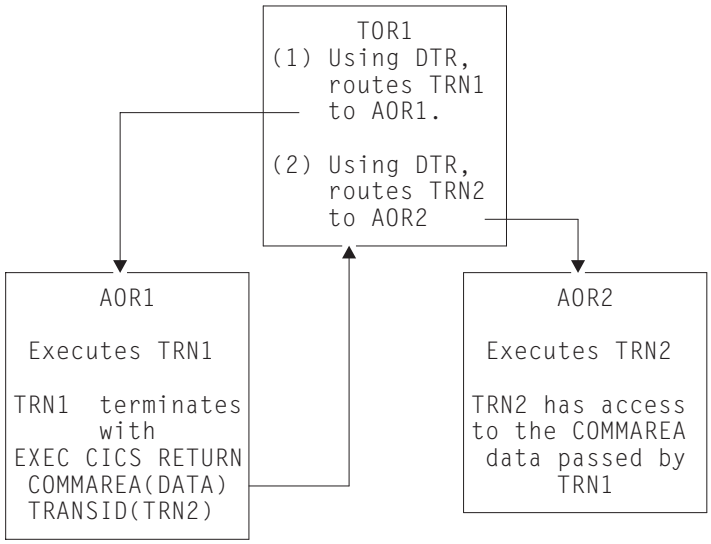


Figure 89. The use of a COMMAREA by a pseudoconversation in a dynamic transaction routing environment

The COMMAREA used in a pseudoconversational transaction, as shown in Figure 89, 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 attribute 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 90 on page 318. 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.

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.

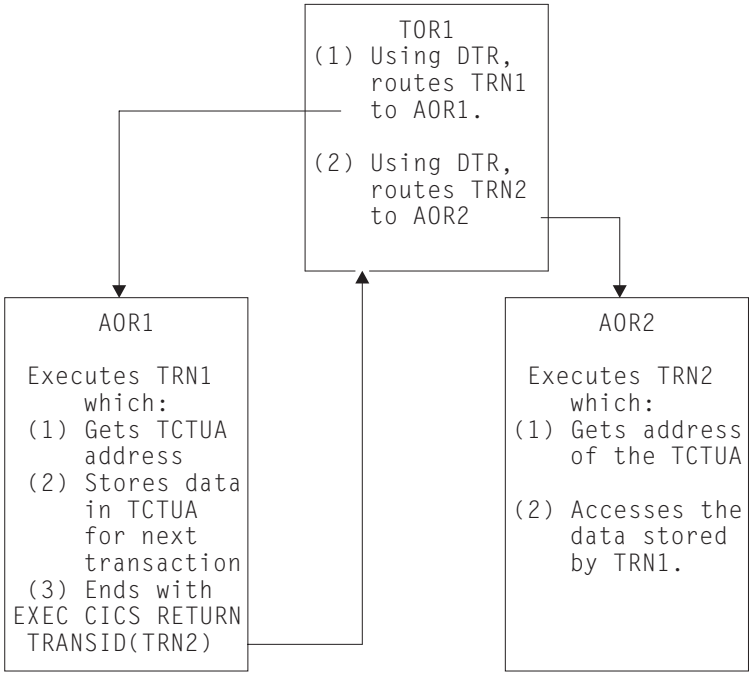


Figure 90. The use of a TCTUA by a pseudoconversation in a dynamic routing environment

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 a 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, if the ENQs and DEQs are supported by appropriate ENQMODEL resource definitions (see ENQMODEL resource definitions in 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 determines 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. 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 ENQ|DEQ 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 remains blank (the default value), CICS treats the ENQ|DEQ as local to the issuing CICS region.
  - If the ENQSCOPE is non-blank, CICS treats the ENQ|DEQ 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 ENQ|DEQ command, and the queue name is made up by prefixing the four-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 by defining appropriate ENQMODELs, without any change to the application programs.

Benefits of sysplex enqueue

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 lowering the Systems Management cost of using 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 and can be used to pass data between BTS activities or between different activations of the same activity.

Containers cannot be used outside of an activity; for more information, see the *CICS Business Transaction Services* manual . 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 can 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.

Programming techniques that create affinities

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.

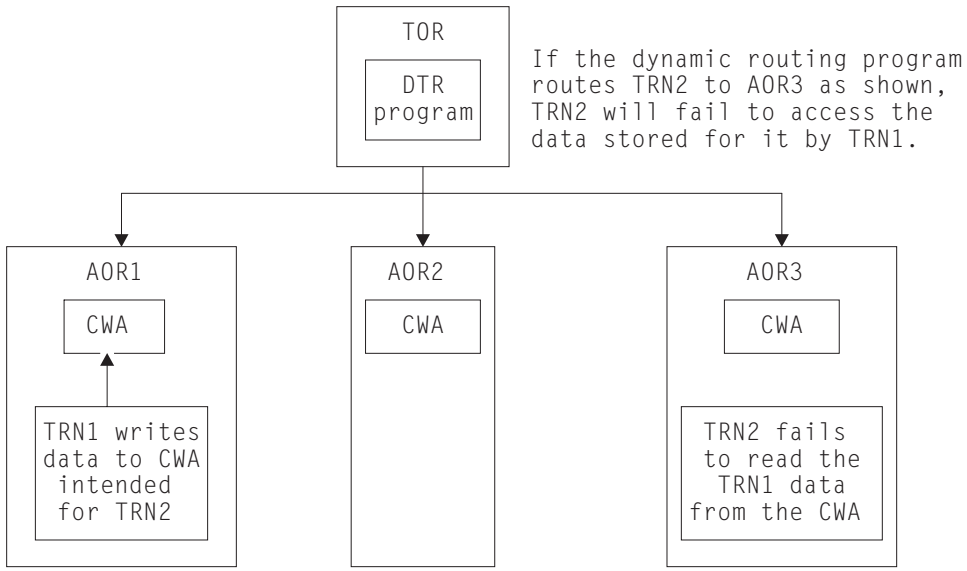
- "Using the common work area" on page 321
- "Using GETMAIN SHARED storage" on page 322
- "Using the LOAD PROGRAM HOLD command" on page 322

- "Sharing task-lifetime storage" on page 324
- "Using the WAIT EVENT command" on page 325
- "Using ENQ and DEQ commands without ENQMODEL resource definitions" on page 326

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 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 91.



CWA

Figure 91. 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 that 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 92 illustrates the use of shared storage.

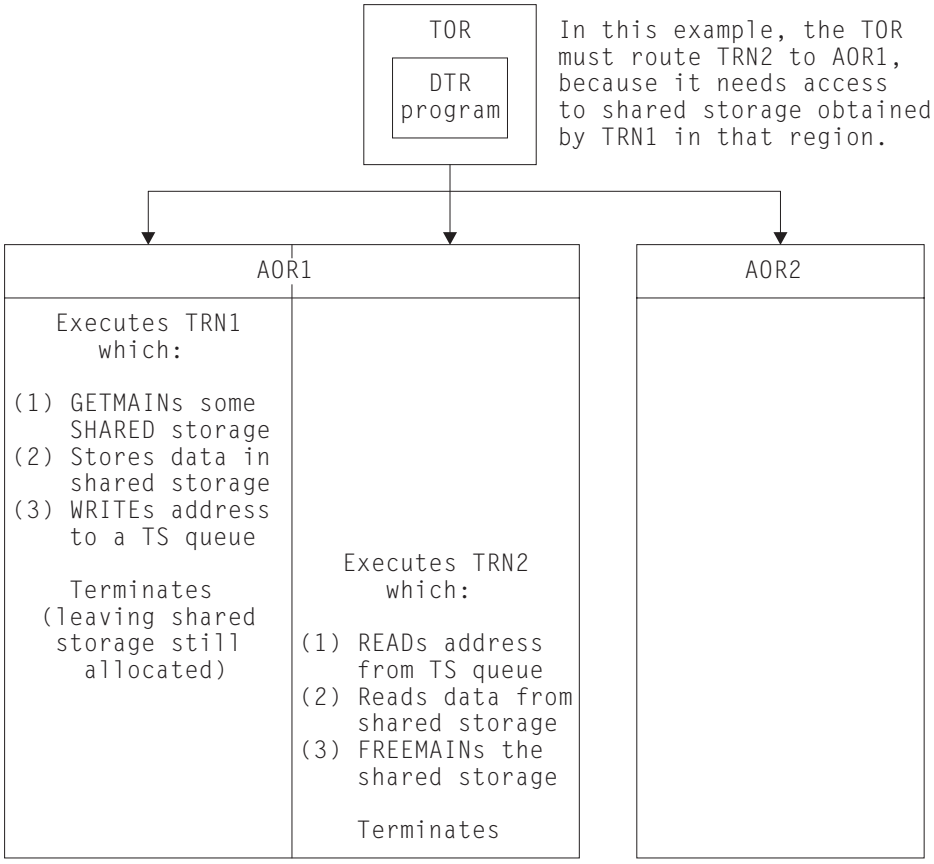


Figure 92. Illustration of inter-transaction affinity created by use of shared storage. The dynamic transaction routing program needs to be aware of this affinity, and ensure that it routes TRN2 to the same target region as TRN1.

If the two transactions shown in Figure 92 are parts of a pseudoconversational transaction, the use of shared storage should be replaced by a COMMAREA (if 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, if:

- 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 typical 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 identical to that caused by the use of GETMAIN SHARED storage (see Figure 92 on page 322 and Figure 93), 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.

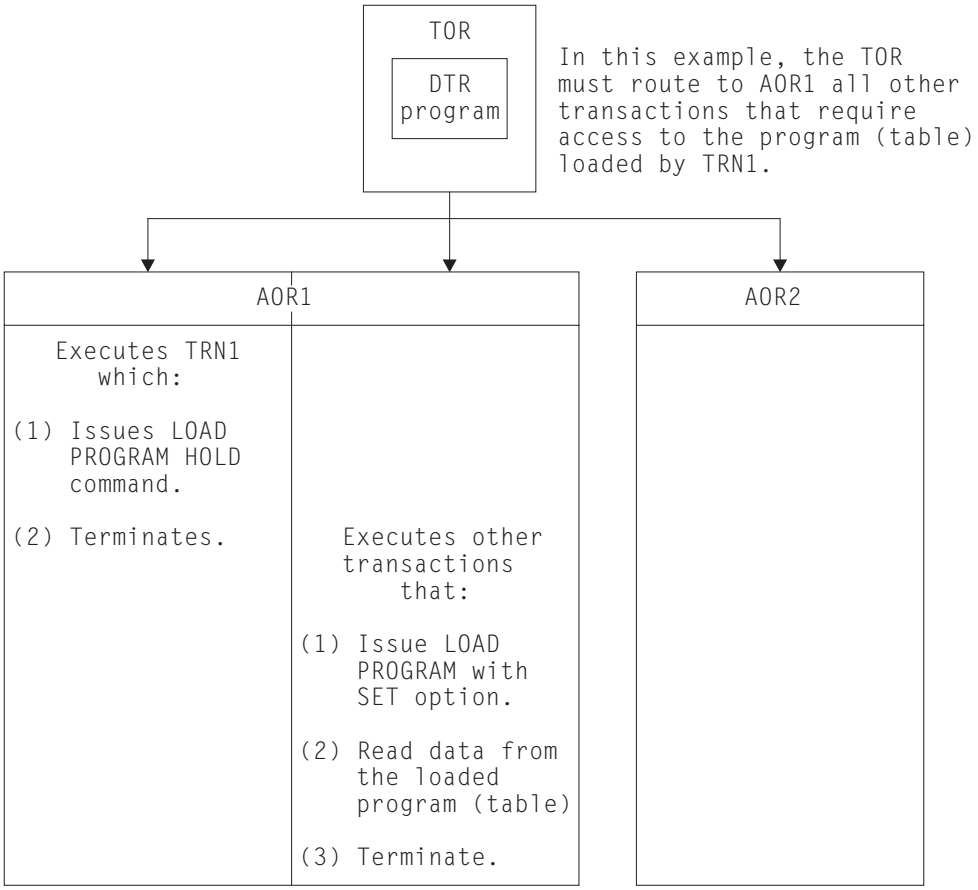


Figure 93. Illustration of inter-transaction affinity created by use of LOAD PROGRAM HOLD. The dynamic routing program needs to be aware of this affinity, and ensure that it routes TRN2 to the same target region as TRN1.

Note: This rule applies also to programs defined with the RESIDENT option on the resource definition for the loaded program (whether 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 94 for an example).

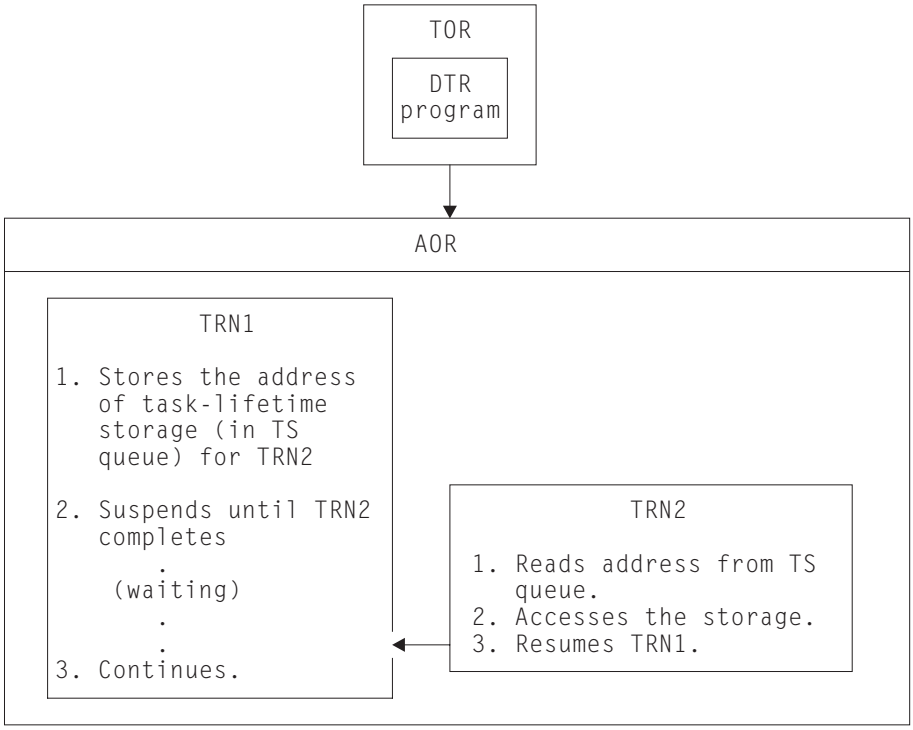


Figure 94. Illustration of inter-transaction affinity created by use of task-lifetime storage. TRN2 must execute in the same target region as TRN1. Also, TRN1 must not terminate until TRN2 has finished using its task-lifetime storage.

For two tasks to share task-lifetime storage belonging to one of them requires that the tasks are synchronized in some way. See Table 25 for commands that provide ways of suspending and resuming a task that passes the address of its local storage.

Table 25. 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. 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 25 on page 324 are fundamentally unsafe because it is possible that the storage-owning task could be purged.

Note:

1. Using synchronization techniques, such as RETRIEVE WAIT/START, to allow sharing of task-lifetime storage 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 95.

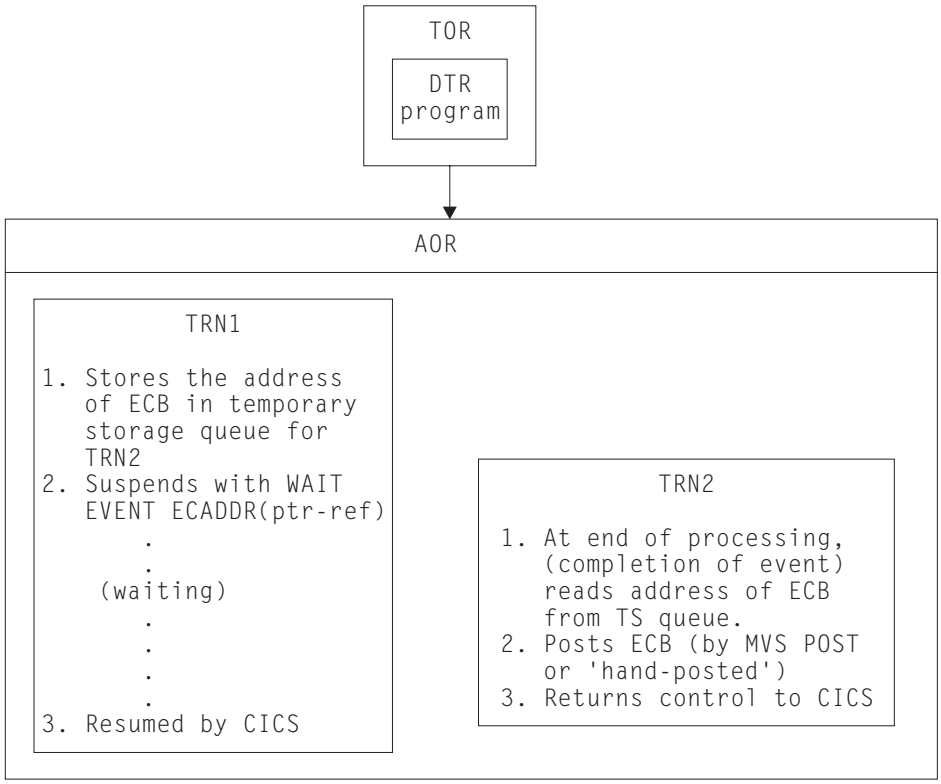


Figure 95. Illustration of inter-transaction affinity created by use of WAIT EXTERNAL command. TRN2 must execute in the same target region as TRN1.

If TRN2 shown in Figure 95 on page 325 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.

The ENQ and DEQ commands 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 319 and ENQMODEL resource definitions in 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 96.

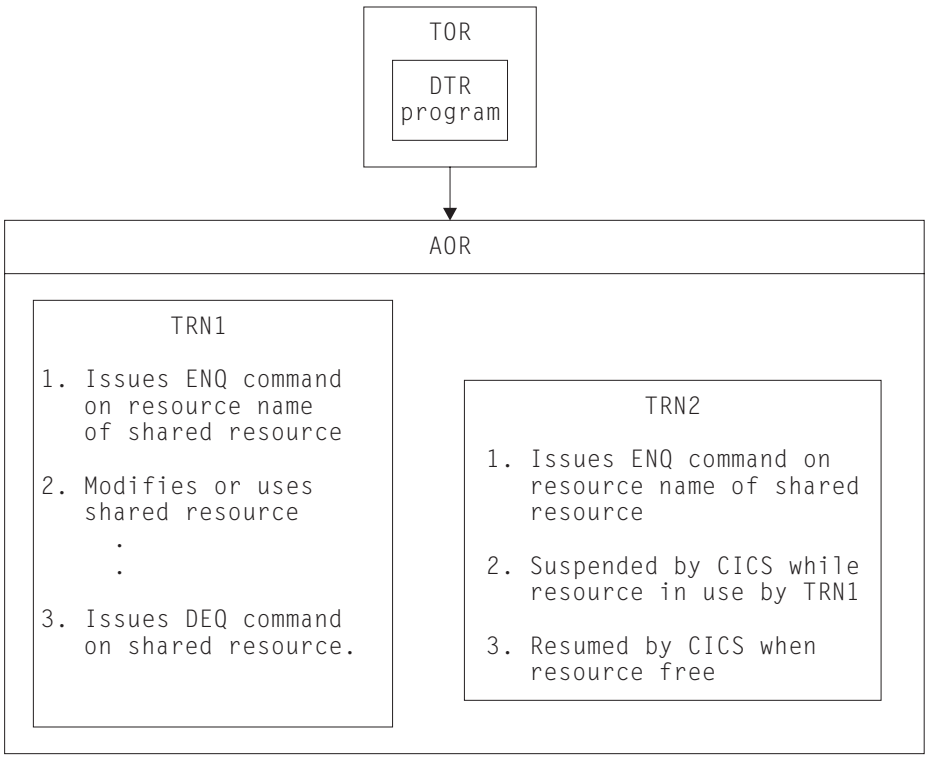


Figure 96. Illustration of inter-transaction affinity created by use of ENQ/DEQ commands. TRN2 must execute in the same target region as TRN1.

If TRN2 shown in Figure 96 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.

Programming techniques that might create affinities

Some CICS application programming techniques might create an affinity between transactions depending on how they are implemented. You can adopt various approaches to avoid unwanted affinities.

- "Avoiding affinities when using temporary storage"
- "Using transient data" on page 329
- "Avoiding affinities when using the RETRIEVE WAIT and START commands" on page 330
- "Avoiding affinities when using the START and CANCEL REQID commands" on page 332
- "Avoiding affinities using the DELAY and CANCEL REQID commands" on page 334
- "Avoiding affinities using the POST and CANCEL REQID commands" on page 336

Avoiding affinities when using temporary storage

CICS application programs commonly use temporary storage queues to hold temporary application data, and to act as scratch pads.

Sometimes a temporary storage 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 temporary storage 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 preferred method for passing data to a DPL program is by a channel or COMMAREA, but if a temporary storage queue is used for passing data in a DPL application, the queue must be shared between the two regions.

Sometimes a temporary storage queue holds information that is specific to the target region, or holds read-only data. In this case the temporary storage queue can be replicated in each target region, and no sharing of data between target regions is necessary.

However, in many cases a temporary storage 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. To make a temporary storage queue globally accessible while avoiding inter-transaction affinities, you can either use remote queues in a queue-owning region (QOR), or use shared temporary storage queues that reside in a temporary storage data-sharing pool in a coupling facility.

Remote temporary storage queues in a queue-owning region

To function ship temporary storage requests to a queue-owning region (QOR):

- In the application-owning regions, create TSMODEL resources using the REMOTESYSTEM and REMOTEPREFIX (or XREMOTEPFX) attributes to define the temporary storage queues as remote.
- In the queue-owning region, create corresponding TSMODEL resources to define the characteristics of the temporary storage queues.
- If you want to avoid duplication, you can create TSMODEL resources that can be installed in both a local and remote region, and share these resources between the application-owning regions and the queue-owning region.

If your temporary storage queue names are generated dynamically, you must follow a process that generates queue names that are unique for a given terminal. The usual convention is a 4-character prefix (for example, the transaction identifier) followed by a 4-character terminal identifier as the suffix. Generic queue names in this format can be defined as remote queues. If your naming convention 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.

You cannot use the global user exits for temporary storage requests to change a temporary storage queue name from a local to a remote queue name.

Figure 97 illustrates the use of a remote queue-owning region.

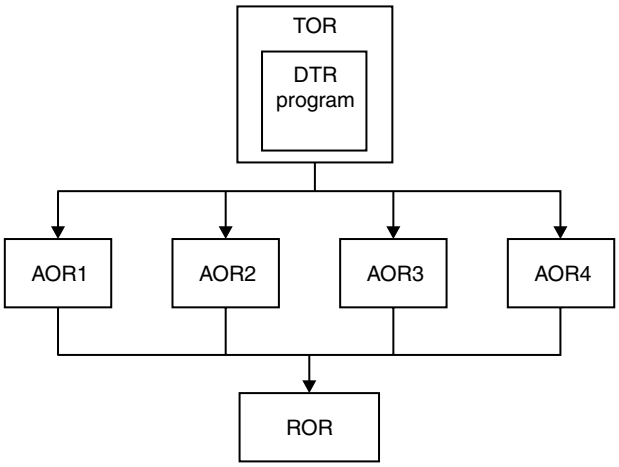


Figure 97. Using remote queues to avoid inter-transaction affinity relating to temporary storage. This example shows a combined file-owning and queue-owning region (a resource-owning region, or ROR). Separate regions can be used, but these require special care during recovery operations because of 'indoubt' windows that can occur when recovering data managed independently by file control and temporary storage control.

Shared temporary storage queues

To use shared temporary storage queues that reside in a temporary storage data-sharing pool in a coupling facility, in the application-owning regions, create TSMODEL resources that use the POOLNAME and PREFIX (or XPREFIX) attributes to map the temporary storage requests to a named pool of shared temporary storage queues.

You must set up a temporary storage server to support each pool of temporary storage queues that is defined in the coupling facility. For more information, see Setting up and running a temporary storage server in Configuring.

Note that TSMODEL resource definitions do not support applications specifying an explicit SYSID for a queue that resides in a temporary storage data sharing pool. Applications must use the QUEUE or QNAME option instead. If your applications use an explicit SYSID for a shared queue pool, this requires the support of a temporary storage table (TST).

Figure 98 shows four AORs that are using the same pool of shared temporary storage queues in a coupling facility. A temporary storage server is always managed by the queue server region program, DFHXQMN.

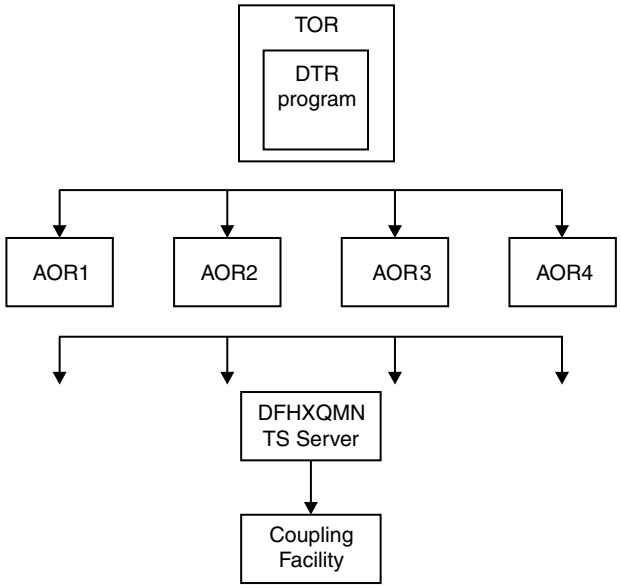


Figure 98. Example of the use of a temporary storage data-sharing server

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. 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

CICS application programs commonly use transient data queues (TD). To enable transactions that use a TD queue that must be shared, to be dynamically routed to a target region, you must ensure that the TD queues are globally accessible.

The dynamic transaction routing considerations for TD queues have much in common with those for temporary storage.

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 99 illustrates the use of a remote transient data queue-owning region.

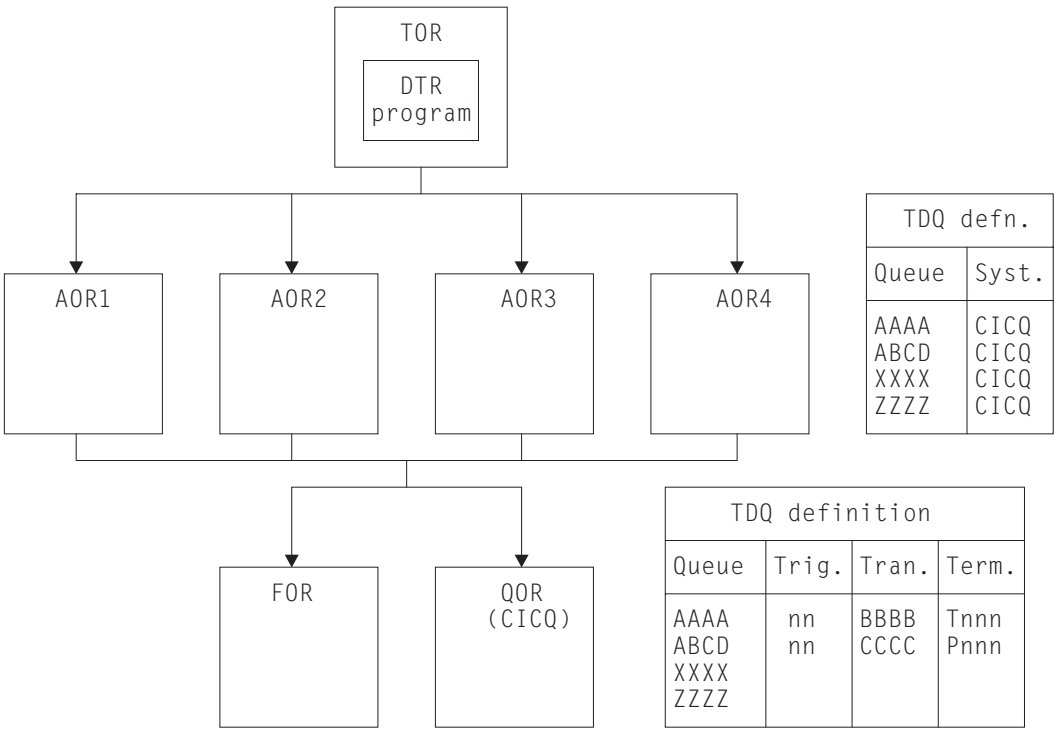


Figure 99. Using remote queues to avoid inter-transaction affinity relating to transient data. The transient data queue definitions installed in the target regions are defined as owned by the QOR (CICQ). All the transient data queue definitions installed in the QOR are local, some with trigger levels.

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).

Avoiding affinities when using the RETRIEVE WAIT and START commands

The use of some synchronization techniques permit the sharing of task-lifetime storage between two synchronized tasks. Use the RETRIEVE WAIT and START commands for this purpose.

In the example illustrated in Figure 100, 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.

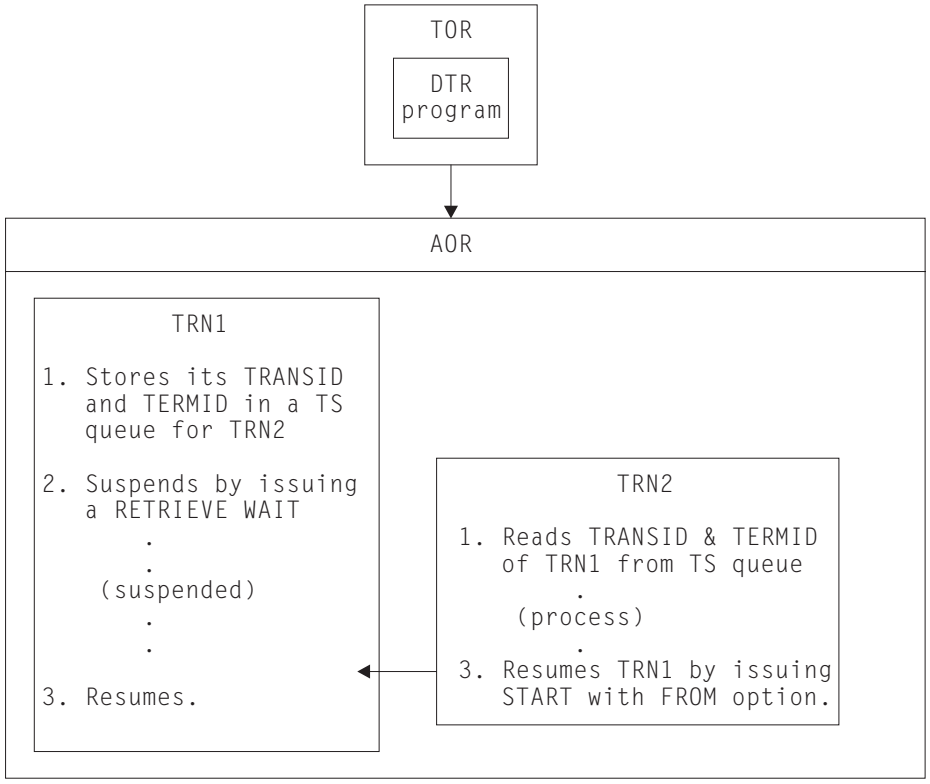


Figure 100. Illustration of task synchronization using RETRIEVE WAIT and START commands

In the example of task synchronization using RETRIEVE WAIT and START commands shown in Figure 100, 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 must 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 must ensure that TRN2 must execute in the same region as TRN1 to preserve the application design.

Avoiding affinities when using the START and CANCEL REQID commands

The CICS START command is used by a transaction to start another transaction. Another transaction can cancel this command using the CANCEL command and specifying the REQID associated with the START command.

With 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.

A temporary storage queue is one way that the REQID can be passed from task to task.

Note: The CANCEL command is effective only if the specified interval has not yet expired. 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 101 on page 333 illustrates this programming technique.

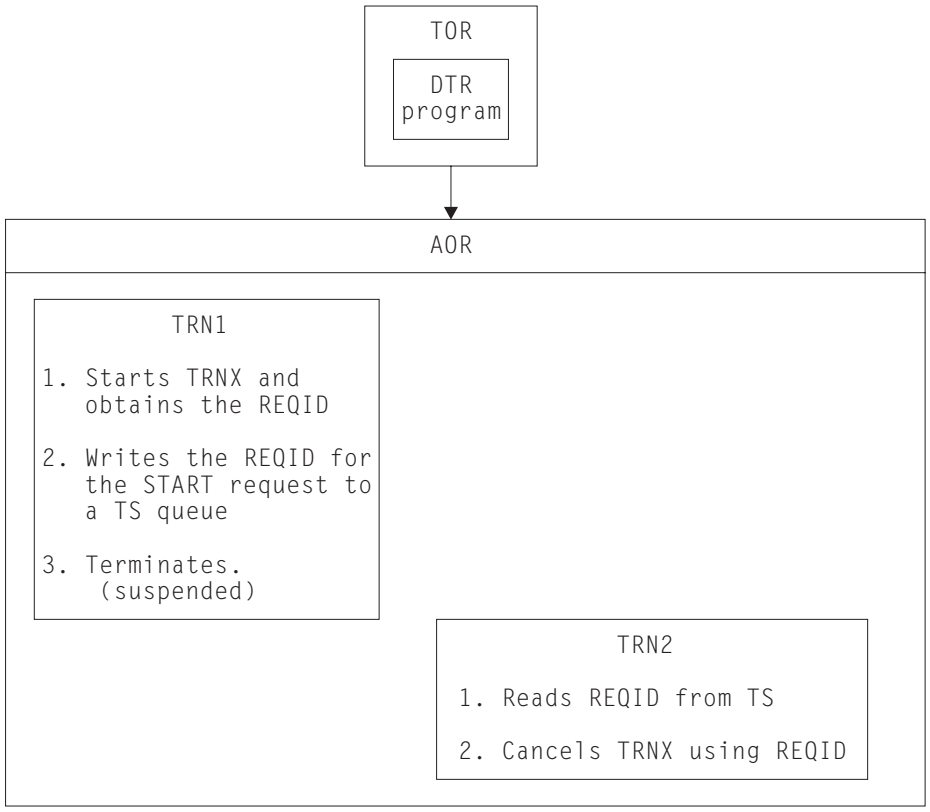


Figure 101. Illustration of the use of the START and CANCEL REQID commands

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 previous 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.

Avoiding affinities using the DELAY and CANCEL REQID commands

The CICS DELAY command is used to suspend a task. Another task can cancel this command using the CANCEL command and specifying the REQID associated with the DELAY command.

With this CICS application programming technique, one task can resume another task that has been suspended by a DELAY command.

A DELAY request can be canceled only 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 canceling the DELAY request.

The process using a TS queue is illustrated in Figure 102 on page 335.

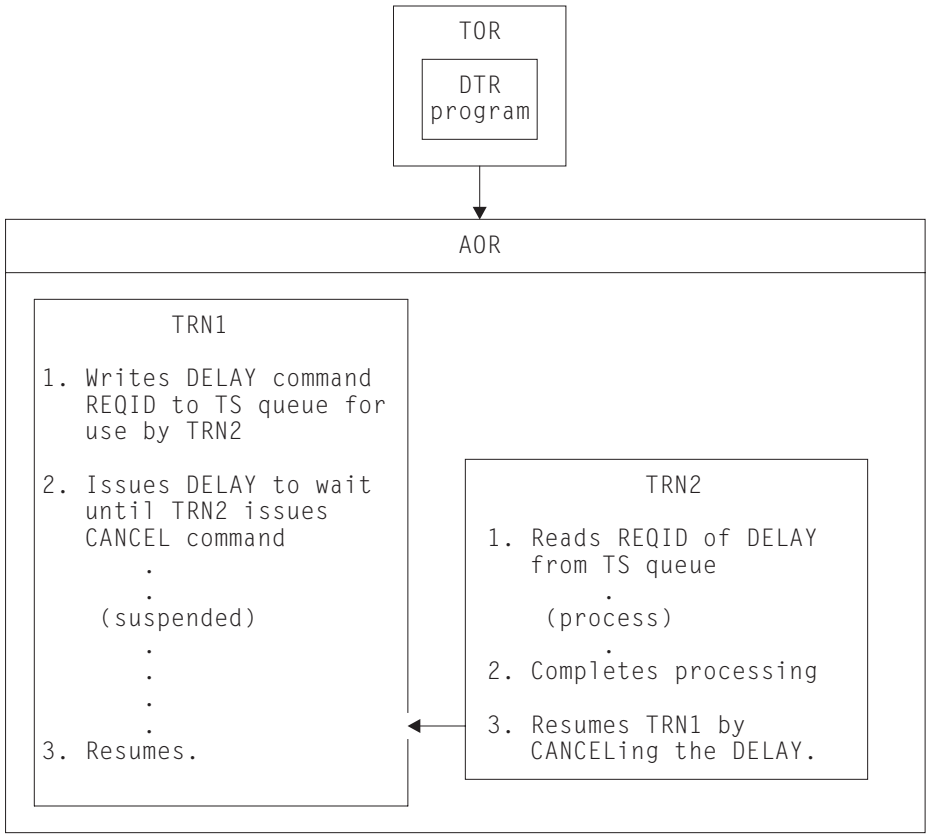


Figure 102. Illustration of the use of the DELAY and CANCEL REQID commands

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.

Avoiding affinities 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) either tests the ECB periodically, or issues a WAIT command on the ECB.

A TS storage queue can be used to pass the REQID of the POST request from task to task.

Figure 103 illustrates this technique.

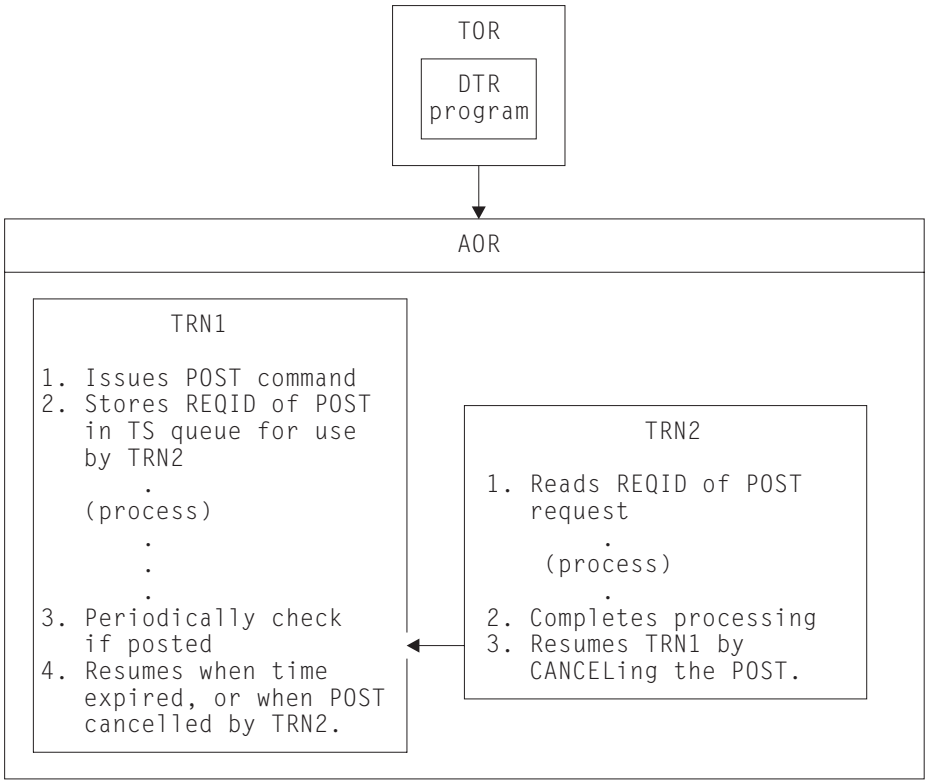


Figure 103. Illustration of the use of the POST command

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. Use the CICS Interdependency Analyzer to detect affinities.

About this task

For a description of the utility and how to use it, see CICS Interdependency Analyzer.

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 CICS Interdependency Analyzer, you can use one of the following methods to detect affinities:

- Use the CICS-supplied load library scanner, CAULMS.

  Note: From CICS TS for z/OS, Version 3.2 the detector and reporter components previously provided as part of the CICS Transaction Affinities utility are not inlcuded in CICS TS. These components, as well as the load library scanner component, are incorporated in the CICS Interdependency Analyzer, which can analyze both interdependencies and affinities. The load library scanner alone remains in CICS TS for z/OS, Version 3.2, and can produce reports on application programs which have potential affinities.

  For details of how to use the CICS load library scanner, see the CICS TS 2.3 *Transaction Utilities Guide*.
- 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 cause inter-transaction affinity.
- Run a trace analysis program that can analyze CICS auxiliary trace. For example, if you run the CICS trace utility program, DFHTU680, 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 can cause difficult problems of inter-transaction affinity:

- The affinity can be hidden from the application programmer.
- The application generator might 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 or distributed 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 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.

This section covers the following topics:

- "Affinity transaction groups"
- "Affinity relations and affinity lifetimes"

Affinity transaction groups

To manage affinities in a dynamic routing environment, you must first categorize transactions by their affinity. One way to do this is to place transactions in transaction groups, where a group is a set of transactions that have inter-transaction affinity.

Each affinity transaction group (or affinity group) 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.

The more inter-transaction affinity you have in a 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 in an affinity group are characterized by their affinity relation and affinity lifetime. These relation and lifetime attributes determine the scope and duration of an affinity. Therefore 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.

Affinity relations and affinity lifetimes

When you create a transaction group, you can assign to the group an affinity relation using the **AFFINITY** attribute and an affinity lifetime using the **AFFLIFE**

attribute. The affinity relation determines how the dynamic or distributed routing program selects a target region for a transaction instance associated with the affinity. The affinity lifetime determines when the affinity ends.

The five possible affinity relations that you can assign to your affinity groups are `BAPPL`, `Global`, `LOCKED`, `LUname`, and `userid`. The affinity lifetime you assign to an affinity depends on the affinity relation. For example, if the affinity relation is `LOCKED`, the affinity lifetime must be UOW, whereas if the affinity relation is BAPPL, you can set the affinity lifetime to `Activity`, `Permanent`, `Process` or `System`.

The BAPPL relation:

When you define an affinity relation of `BAPPL` to a transaction group, all instances of transactions that are associated with the same BTS process must run in the same target region for the lifetime of the affinity.

If the affinity relation is `BAPPL`, the affinity lifetime must have one of the following values:

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.

Figure 104 on page 340 shows an example of an affinity group with the BAPPL relation.

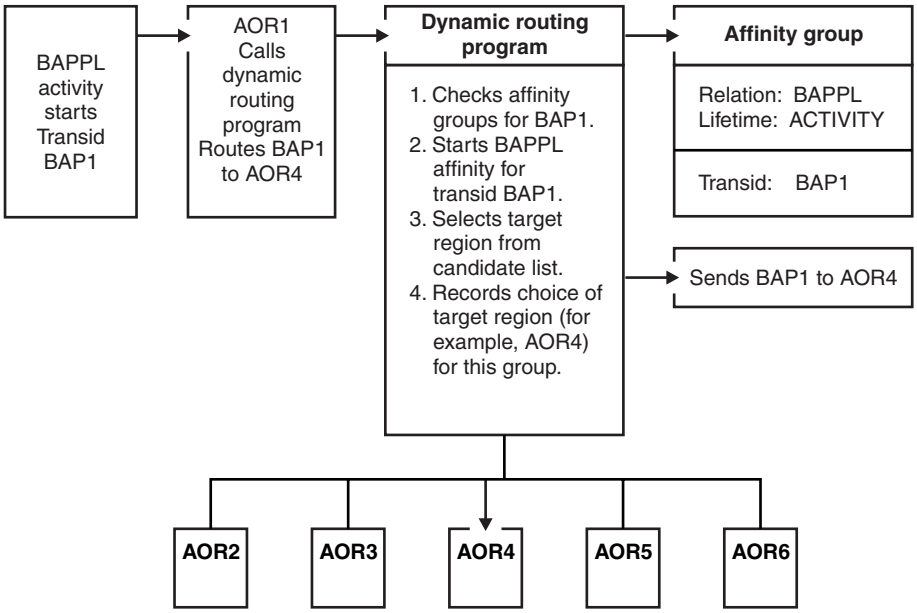


Figure 104. Managing inter-transaction affinity with BAPPL relation and activity lifetime

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 might introduce affinities. You must define such affinities to workload management. It is particularly important to specify each affinity's lifetime. Failure to do this might 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 able to honour only 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. These affinities 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 lifetime of PERMANENT. If a total system failure or a planned shutdown occurs, affinities are lost, but activities in CICS will be recovered from the BTS RLS data set.

The global relation:

When you define an affinity relation of `global` to a transaction group, all instances of all the transactions started from any terminal, by any START command, or by any CICS BTS process, must run in the same target region for the lifetime of the affinity.

If the affinity relation is `global`, the affinity lifetime must have one of the following values:

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 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.

Figure 105 shows an example of a global relation.

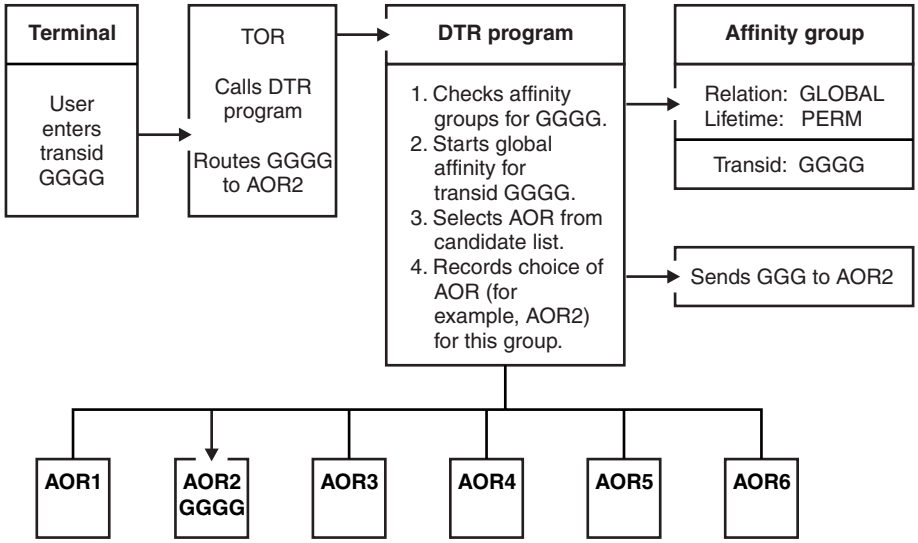


Figure 105. Managing inter-transaction affinity with global relation and permanent lifetime

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 LOCKED relation:

When you define an affinity relation of `LOCKED` to a transaction group, all instances of transactions in the group that are associated with dynamically-linked programs that have the same unit of work must run in the same target region for the lifetime of the unit of work.

The LOCKED relation is associated with a unit of work (UOW). The affinity lifetime for the LOCKED relation is always UOW. The unit of work ends either when a CICS **SYNCPOINT** or **ROLLBACK** request is run, or when the originating task terminates. UOW is valid only for LOCKED affinities between dynamic program link requests.

A typical example of transactions that have a LOCKED relation is where multiple invocations of the same dynamically-routed program access a common resource. Each invocation of the program has a transaction code under which the called program runs on a target system. By specifying that transaction code as part of a transaction group that designates a LOCKED affinity, all instances of dynamic program links for that program name cause the associated remote transaction to be rerouted to the same target region until the caller's unit of work ends. If those subsequent program links were routed to different regions, deadlocks could occur. Therefore work must not be routed away from the region that locked the resource. For further information, see Multiple links to the same server region, in the *CICS Intercommunication Guide*.

Figure 106 shows an example of an affinity with the LOCKED relation.

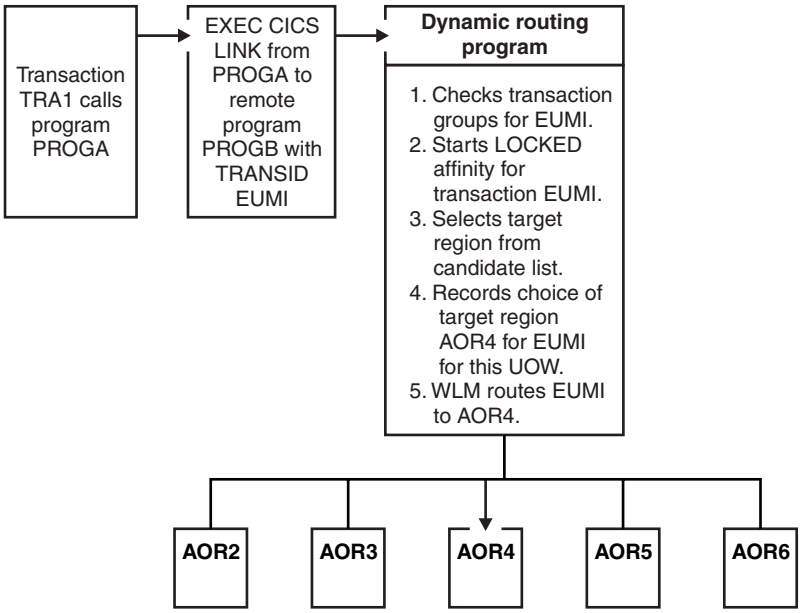


Figure 106. Managing inter-transaction affinity with LOCKED relation and UOW lifetime

In this example, transaction TRA1 calls program PROGA as part of a unit of work identified by the **NETUOWID** attribute of the WLMATAFF resource table. PROGA makes an **EXEC CICS LINK** request to program PROGB on a remote system. PROGB is defined as dynamic and its associated transaction, defined by the **TRANSID** attribute in the program definition, is EUMI. EUMI is a user-defined mirror transaction belonging to a transaction group that is defined with an affinity relation of LOCKED and an affinity lifetime of UOW. Workload management routes this transaction to CICS region AOR4. The dynamic routing program, after checking whether an affinity already exists, starts a new affinity for EUMI. All subsequent instances of EUMI share this affinity for the lifetime of the unit of work, and are routed to AOR4 until the unit of work ends.

The LUname (terminal) relation:

When you define an affinity relation of LUname to a transaction group, all instances of all the transactions in the group that are associated with the same terminal must run in the same target region for the lifetime of the affinity.

If the affinity relation is LUname, the affinity lifetime must have one of the following values:

Pseudoconversation
The affinity lasts for the whole pseudoconversation, and ends when the pseudoconversation ends at the terminal. Each transaction must end with 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 LUname relation are those that:

- Use a local TS queue to pass data between the transactions in a pseudoconversation, and
- Use a TS queue name that is derived, in part, from the terminal name (see "Avoiding affinities when using temporary storage" on page 327 for information about TS queue names).

These types of transactions can be placed in an affinity group with a relation of terminal and a lifetime of pseudoconversation. When the dynamic routing program detects the first transaction in the pseudoconversation initiated by a specific terminal (LUname), it is free to route the transaction to any target region that is a valid candidate for that transaction. However, any subsequent transaction in 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 might occur in many CICSplexes. It can be managed easily by a dynamic routing program, and should not inhibit the use of dynamic routing.

Figure 107 on page 344 shows this type of example.

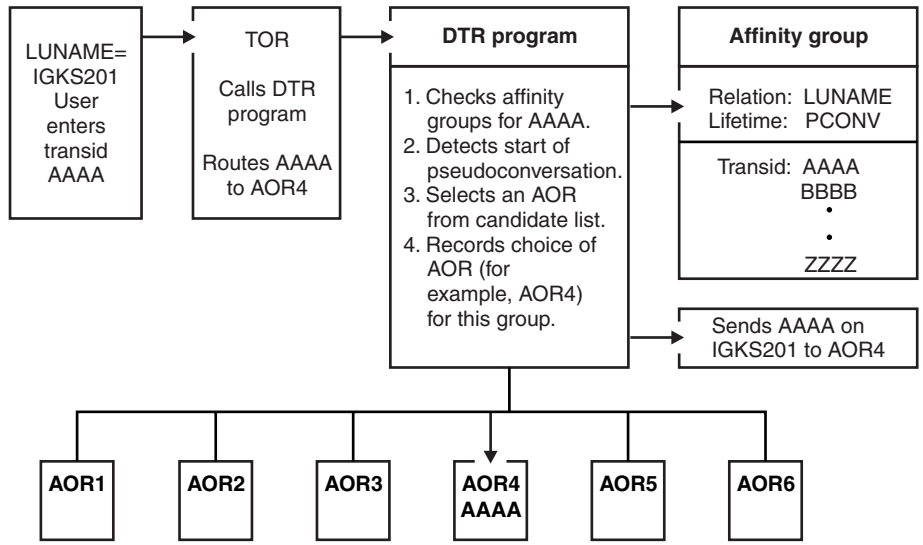


Figure 107. Managing inter-transaction affinity with LUname relation and pseudoconversation lifetime

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:

When you define an affinity relation of `userid` to a transaction group, 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 user ID, must execute in the same target region for the lifetime of the affinity.

If the affinity relation is `userid`, the affinity lifetime must have one of the following values:

Pseudoconversation
The affinity lasts for the whole pseudoconversation, and ends when the pseudoconversation ends for that user ID. Each transaction must end with **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. This lifetime is possible only in those situations where only one user per user ID is permitted. Signon lifetime cannot be detected if multiple users are permitted to be signed on with the same user ID 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 user ID 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.

Figure 108 shows an example of an affinity group with the userid relation and a signon lifetime.

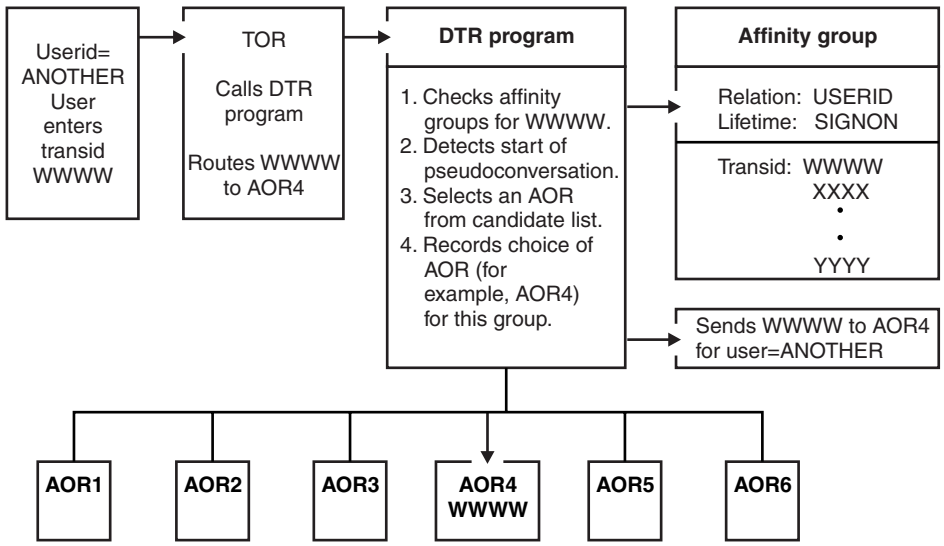


Figure 108. Managing inter-transaction affinity with `userid` relation and `signon` lifetime

In this example, any instance of a transaction from a terminal starts a `signon` 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.

Recovery design

CICS provides two techniques - journaling and syncpointing - that can help you to recover or reconstruct events or data changes during CICS execution.

Techniques for named counter recovery are described in "Named counter recovery" on page 653.

Journaling

CICS provides facilities for creating and managing *journals* during CICS processing. Journals can contain any 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 "Dealing with exception conditions" on page 349. (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 can 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 that 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

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 this command, 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 can 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 previously listed events 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 indoubt tasks only (high-order bit off)
- For all records encountered until the scan is terminated (high-order bit on)

For more information about journal records, see Journal records. For more information about the recovery process, see CICS emergency restart in the *CICS Recovery and Restart Guide*.

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 Starting CICS with the START=INITIAL parameter in 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 Recovery and restart facilities in 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 want 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.

Restriction: 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.)

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" in “Examples of distributed program link” on page 374 and the *CICS Application Programming Reference*.

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.

Consult your system programmer if syncpoints are to be issued in a transaction that is eligible for transaction restart.

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 that the CICS EXEC interface program passes this condition back to your application. This condition is also called a RESP value, because you can obtain it by using the RESP option in your command. Alternatively, you can obtain this value by reading it from the EXEC interface block (EIB). The condition is usually NORMAL.

If something unusual happens, you get an *exception condition*, that is, 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 value (RESP2) associated with them, which gives further information. To obtain this RESP2 value, you can use the RESP2 option (in addition to the RESP option) in your command, or you can read it from the EIB.

Some conditions, even though they are not NORMAL, do not indicate an error situation. For example, an ENDFILE condition on a READNEXT command during a file browse, might indicate expected behavior. For information about all possible conditions and the commands on which they can occur, see Application development reference.

This section describes:

- "Default CICS exception handling"
- "Handling exception conditions by inline code" on page 351
- "Modifying default CICS exception handling" on page 354

Default CICS exception handling

For COBOL, PL/I, and assembler language applications (but not AMODE(64) assembler language applications), unless you specify otherwise, CICS uses its built-in exception handling whenever an exception condition occurs. For AMODE(64) assembler language applications and applications written in C or C++, CICS takes no action when an exception condition occurs. The application must handle the exception condition.

See "Handling exception conditions by inline code" on page 351 for information about handling exception conditions.

With CICS default exception handling, when an exception condition occurs, the most frequent action is to cause an abend. For details about the specific behaviors for each condition and for each command, refer to Application development reference and Introduction to System programming commands in Developing system programs.

If the CICS default exception handling does not meet your requirements, you can specify other actions in the following ways:

- Turn off the CICS default exception handling on a specific EXEC CICS command call by specifying the NOHANDLE option.
- Turn off the CICS default exception handling by specifying the RESP option on the command. This option switches off the default CICS exception handling in the same way as the NOHANDLE command. It also updates the variable named by the argument of the RESP option with the value of the condition returned by the command. For details, see "Handling exception conditions by inline code" on page 351.

If you turn off the default CICS exception handling, you must ensure that your program copes with anything that might happen in the command call.

It is possible to use combinations of the HANDLE ABEND, HANDLE CONDITION, and IGNORE CONDITION commands to modify the default CICS exception handling, but this is no longer recommended. For details, see "Modifying default CICS exception handling" on page 354.

Handling exception conditions by inline code

If your program is written in C or C++, or is an AMODE(64) assembler language application, inline code is the only technique available to handle exception conditions. If your program is not written in C or C++ and is not an AMODE(64) assembler language application, handling exception conditions involves either using the NOHANDLE option or specifying the RESP option on **EXEC CICS** commands, which prevents CICS from performing its default exception handling.

About this task

The RESP option makes the value of the exception condition directly available to your program, for it to take remedial action.

If you use the NOHANDLE or RESP option, ensure that your program can cope with whatever condition might 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 might 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. For more information about EIB, see EIB fields in Reference -> Application development. 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 easy to test the RESP value, because you can examine RESP values symbolically. This technique 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 that corresponds to the condition that might have been raised. Normally, its value is DFHRESP(NORMAL).

Use of RESP and DFHRESP in COBOL and PL/I

The following example shows an EXEC CICS call in COBOL that uses the RESP option. A PL/I example is similar, but ends with a semicolon (;) rather than END-EXEC.

```
EXEC CICS WRITEQ TS FROM(abc)
          QUEUE(qname)
          NOSUSPEND
          RESP(xxx)
          END-EXEC.
```

The following code is an example of using DFHRESP to test for the RESP value:

```
IF xxx=DFHRESP(NOSPACE) THEN ...
```

Use of RESP and DFHRESP in C and C++

The following example shows an EXEC CICS call in C that uses the RESP option, including the declaration of the RESP variable:

```
long response;
:
EXEC CICS WRITEQ TS FROM(abc)
          QUEUE(qname)
          NOSUSPEND
          RESP(response);
```

The following code is an example of using DFHRESP to test for the RESP value:

```
if (response == DFHRESP(NOSPACE))
{
:
}
```

Use of DFHRESP in assembler language

The following code is an example of a test for the RESP value in assembler language:

```
  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 109. 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 EXEC CICS statement asks for a map of the name given by *mapname*, of the mapset ACCOUNT, to be read into the area of memory to which the variable *map* points, with the value of the condition being held by the variable *response*.

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.

Specific cases for two conditions:

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.

In this example, any other condition is held to be an error. The example sets ExitKey to DFHCLEAR—the same value that it would have set if the user had cleared the screen—which it then returns to the calling program. By checking the return code from ReadAccountMap, the calling program would know that the map had not been updated and that some remedial action is required.

An example of exception handling in COBOL

The following example is a typical function which could be used to receive a BMS map and to cope with exception conditions.

```
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.

MODIFYMAP.
```

Figure 110. An example of exception handling in COBOL

MAPNAME is the variable which contains the name of the map which is to be received.

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

You can use the **HANDLE CONDITION**, **IGNORE CONDITION**, and **HANDLE ABEND** commands in your application program to modify how CICS handles exceptions and abends.

About this task

This information applies only to COBOL, PL/I, or assembler language applications (but not AMODE(64) assembler language applications). For any other supported high level language, these commands are not supported.

A summary of the commands is as follows:

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 most frequent way in which CICS handles exception conditions.

The current effect of the **IGNORE CONDITION**, **HANDLE ABEND**, and **HANDLE CONDITION** commands can be suspended by using the **PUSH HANDLE** command, and reinstated by using the **POP HANDLE** command.

You can pass control to a specified label in the following ways:

- Use a **HANDLE CONDITION** condition(label) command, where condition is the name of an exception condition.
- Use a **HANDLE CONDITION** ERROR(label) command.

The **HANDLE CONDITION** command sets up CICS code to name specific conditions, and then uses this code to pass control to appropriate sections of your application if those conditions arise. With an active **HANDLE CONDITION** command, control goes to the label that you specified for that specific condition.

The same condition might occur on many different commands, and for different reasons. For example, an IOERR condition can occur during file control operations, interval control operations, and others. Therefore, you need to identify the command that has raised a specific condition before you can investigate why it has happened. This is a good reason for using the RESP option in new CICS

applications. Although you need only one **HANDLE CONDITION** command to set error-handling for several conditions, sometimes it is awkward to identify exactly which of several **HANDLE CONDITION** commands is currently active when a CICS command fails somewhere in your code.

If a condition that you have not named occurs, CICS takes the default action, unless this action 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 if the condition occurs, CICS takes the default action.

A common source of errors with the **HANDLE CONDITION** command is not dealing with all conditions. If you use an unfamiliar command, check the command description to find out which exception conditions are possible (see Application development reference). Even when you issue HANDLE commands for all the exception conditions, the error-handling code might not be finished. Sometimes the result is an error-handling routine that, by issuing a RETURN command, allows incomplete or incorrect data changes to be committed.

A good approach is to use the **HANDLE CONDITION** command, but to let the system default action proceed if there is no obvious way round a specific problem.

Bearing in mind the distinction between an error condition, a condition that merely causes a wait (see "How CICS keeps track of what to do" on page 357 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** commands might be useful. See "Using PUSH HANDLE and POP HANDLE commands" on page 360.

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.

About this task

When you use this command, you must include the name of the condition and ensure that the **HANDLE CONDITION** command is executed before the command that might give rise to the associated condition.

Restriction: This command is supported only in COBOL, PL/I, and assembler language applications (but not AMODE(64) assembler language applications). It is not supported in all other supported high level languages.

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 in the same list to specify that all other conditions 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 command remains active while the program is running, or until one of the following events:

- An **IGNORE CONDITION** command for the same condition is met. The **HANDLE CONDITION** command is overridden.
- Another **HANDLE CONDITION** command for the same condition is met. The new command overrides the previous one.

The **HANDLE CONDITION** command 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 301 for information about logical levels.)

The following example shows how to handle conditions, for example DUPREC and LENGERR, that can occur when you use a **WRITE** command to add a record to a data set. In the example, DUPREC is handled as a special case. Standard system action is taken for LENGERR; that is, terminate the task abnormally. All other conditions are handled by the error routine ERRHANDL.

```
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 inline code” on page 351. 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 inline code” on page 351.

   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 355 and “Using the HANDLE CONDITION ERROR command.”
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*.

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

The following example shows you how to trap unexpected errors with the HANDLE CONDITION ERROR command.

About this task

Figure 111 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 111. Trapping the unexpected with the HANDLE CONDITION ERROR command

The HANDLE CONDITION ERROR command passes control to the paragraph at label OTHER-ERRORS if any condition arises for a command that does not specify NOHANDLE or RESP.

This 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. However, 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 raises one of the named conditions.

In this, and the other ACCT programs, you generally use the RESP option. All the commands that specify the RESP option are written with a "catch-all" test (`IF RESPONSE NOT = DFHRESP(NORMAL) GO TO OTHER-ERRORS`) **after** any explicit tests for specific conditions. Therefore, other than the exceptions that you specifically anticipate, any exceptions 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 the same place if they result in any condition other than NORMAL because of this HANDLE CONDITION ERROR command.

Using the IGNORE CONDITION command

You can use the IGNORE CONDITION command to specify that a program continues when a specific condition occurs (in contrast to the HANDLE CONDITION command, which specifies that control passes to a specified label when a specific condition occurs).

About this task

Restriction: This command is supported only in COBOL, PL/I, and assembler language applications (but not AMODE(64) assembler language applications). It is not supported in all other supported high level languages.

You can set 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 that follows the command and return codes are set in the EIB. The following example ignores the MAPFAIL condition:

```
EXEC CICS IGNORE CONDITION MAPFAIL
END-EXEC.
```

When a single EXEC CICS command is processed, it might raise several conditions. For example, a file control command might be invalid and might also apply to a file that is not defined in the file control table. CICS checks these conditions and passes the first one that is not ignored (by your IGNORE CONDITION command) back to your application program. CICS passes back only one exception condition at a time to your application program.

An IGNORE CONDITION command for a given condition applies only to the program you put it in. It remains active while the program is running, or until a later HANDLE CONDITION command that names the same condition is met, in which case the IGNORE CONDITION command is overridden.

You can use an IGNORE CONDITION command for a program that reads records that might be longer than the space you provided, but you do not consider this as 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 the program must check for return codes in the EIB. See "How CICS keeps track of what to do" on page 357 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 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 any additional conditions in further IGNORE CONDITION commands.

Using the HANDLE ABEND command

The **HANDLE ABEND** command activates or reactivates a program-level abend exit in your application program. You can also use this command to cancel a previously activated exit.

For certain programming languages, you can use the HANDLE ABEND command to 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 continue to run. You provide the user exit programs and rely on CICS to call those programs when required.

The flow of control during abend processing is shown in Figure 112 on page 365.

Restrictions

- The **HANDLE ABEND** command does not apply to Java programs.
- Because exception conditions in C and C++ programs do not cause abends, you cannot use the **HANDLE ABEND** command in this way with these programs. However, the **HANDLE ABEND** command is supported in C and C++ when used with the PROGRAM option.
- You cannot use **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.
- You cannot use **HANDLE ABEND LABEL** in AMODE(64) programs.

Using PUSH HANDLE and POP HANDLE commands

The **PUSH HANDLE** and **POP HANDLE** commands suspend and reinstate, respectively, the effects of **HANDLE CONDITION**, **IGNORE CONDITION**, **HANDLE ABEND**, and **HANDLE AID** 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.

Restriction: These commands are supported only in COBOL, PL/I, and assembler language applications (but not AMODE(64) assembler language applications). They are not supported for all other supported high level languages.

CICS also keeps a table of conditions for each **PUSH HANDLE** command that has not been countermanded by a matching **POP HANDLE** command.

When each condition occurs, CICS uses 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.
2. If an entry for the condition exists, this entry determines the action.
3. If no entry exists and the default action for this condition is to suspend execution, the action is as follows:
   a. If the command has the NOSUSPEND or NOQUEUE option, control returns to the next instruction.
   b. If the command does not have the NOSUSPEND or NOQUEUE option, the task is suspended.
4. If no entry exists and the default action for this condition is to abend, CICS searches for the ERROR condition:

a. If the ERROR condition is found, this entry determines the action.
b. If ERROR cannot be found, the task is abended. You can choose to handle abends.

Note: The OVERFLOW condition on a **SEND MAP** command is an exception to the rules.

The **ALLOCATE**, **ENQ**, **GETMAIN**, **WRITE JOURNALNAME**, **WRITE JOURNALNUM**, **READQ TD**, and **WRITEQ TS** commands can all raise conditions for which the default action is to suspend your application program until the specified resource becomes available. Therefore, on these commands, you can use the NOSUSPEND option to inhibit this wait 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: Because 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 function key responses are ignored, and is the reason for testing them earlier in the ACCT code. See "Using the HANDLE AID command" on page 728.

Abnormal termination recovery

CICS provides a program-level abend exit facility so that you can write your own exits (programs or routines) that can receive control during abnormal termination of a task. An example of a function performed by such an abend exit is the cleanup of a program that has started but has not completed normally.

Some causes of abnormal terminations are as follows:

- A user request, for example, by the following code:
  ```
  EXEC CICS ABEND ABCODE(...)
  ```
- A CICS request as a result of an invalid user request. For example, an invalid FREEMAIN request results in the transaction abend code ASCF.
- A program check. The system recovery program (DFHSRP) is driven and the task abends with code ASRA.
- An operating system abend. The DFHSRP program is driven and the task abends with code ASRB.
- A looping task. The DFHSRP program 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 Troubleshooting and support for full details about fixing problems. See 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 in your application program; you can also use this command to cancel a previously activated exit.

When activating an exit, you must do one of the following:

- Use the PROGRAM option to specify the name of a program to receive control.
- For COBOL or assembler applications only (but not AMODE(64) assembler applications), use the LABEL option to specify a routine label to which control branches when an abnormal termination condition occurs.

  You cannot use the **HANDLE ABEND LABEL** command with C, C++, PL/I or AMODE(64) applications. In PL/I, the equivalent is using an ON ERROR block.

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 "Program control" on page 300.)

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 112 on page 365, 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. For programming information about how to customize DFHPEP, see Writing a program error program in Developing system programs.

CICS deactivates the exit upon entry to the exit routine or program to prevent recursive abends in an abend exit. If you want to try the operation again, 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 with the **PUSH HANDLE** and **POP HANDLE** commands, as described in "Using PUSH HANDLE and POP HANDLE commands" on page 360.

When an abend is handled, the dynamic transaction backout program is not invoked. If you need the dynamic transaction backout program, you take an implicit or explicit sync point, 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 sync point processing can 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 section describes:

- "Creating a program-level abend program or routine" on page 363
- "Retrying operations" on page 364
- "Trace" on page 365
- "Monitoring application performance" on page 366
- "Dump" on page 367

Creating a program-level abend program or routine

You can create a program-level abend program. For some programming languages, you can create a program-level abend routine.

Program-level abend program

You can define an abend program 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 might 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.

On entry to an abend exit program, no addressability can be assumed other than that normally assumed for any application program coded in that language.

Program-level abend routine

Restriction: This section does not apply to C, C++, PL/I, and AMODE(64) applications, because the **HANDLE ABEND LABEL** command is not supported.

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.

On entry to an abend exit routine, the register values are as follows:

COBOL
Control returns to the **HANDLE ABEND** command with the registers restored; a COBOL GOTO is then executed.

Assembler (but not AMODE(64) assembler)

Reg 15
Abend label.

Reg 0-14
Contents at the time of the last CICS service request.

Termination of processing

You can terminate processing in an abend exit routine or program in one of the following ways. When abend routines and programs are called by CICS internal logic, they should terminate with an abend because further processing will probably cause more problems.

- Use a **RETURN** command to indicate that the task continues to run 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.

- Use an **ABEND** command to indicate that the task is 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.
- Branch to retry an operation. When you use this method to retry 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 from reentering the abend exit.

For an abend that is 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, issuing a further request for the same service can cause unpredictable results. Because the reinitialization of pointers and work areas, and the freeing of storage areas in the exit routine, might 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.

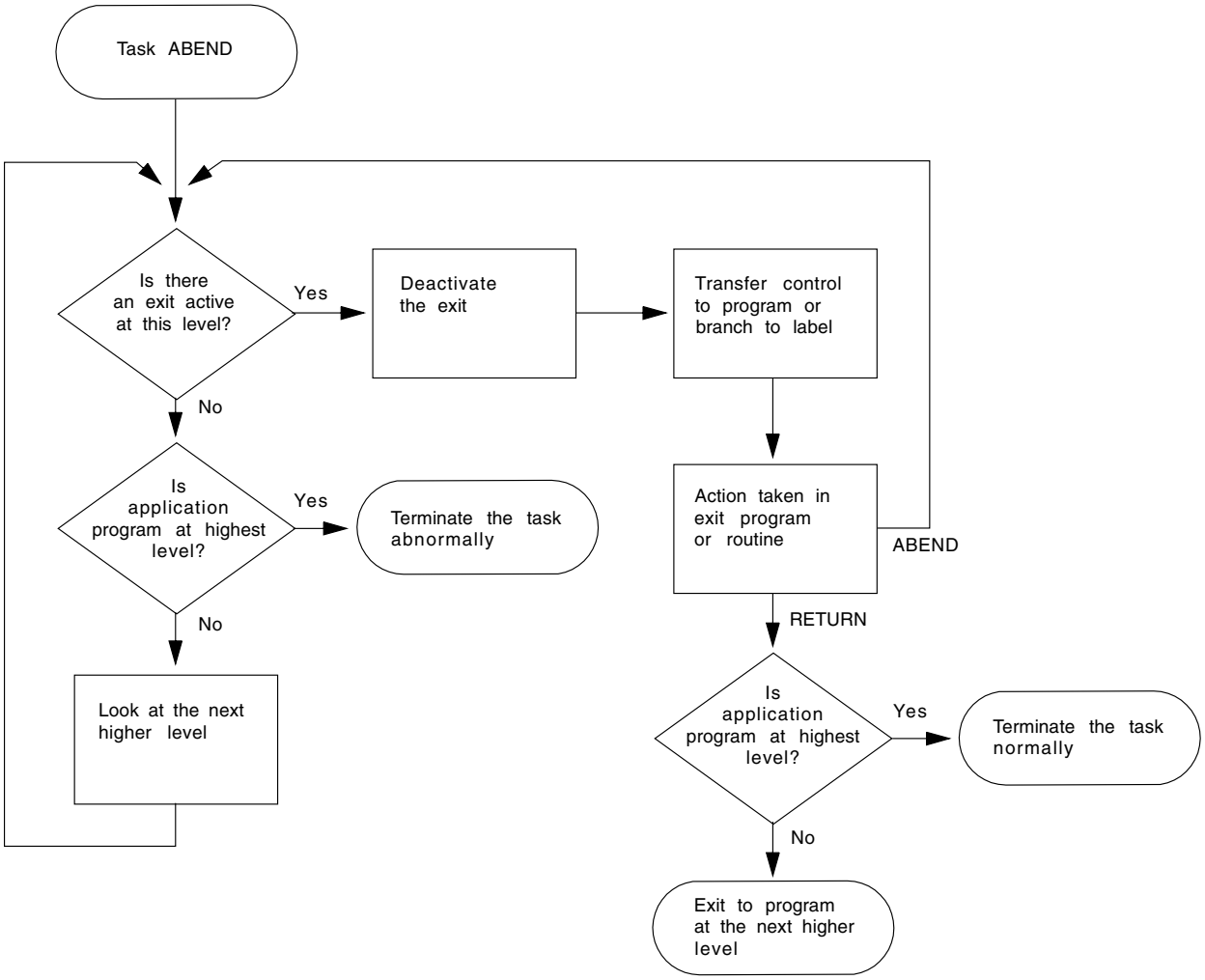


Figure 112. ABEND exit processing

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 Using traces in problem determination in the *CICS Problem Determination Guide*.

You can:

- Specify user trace entry points (ENTER TRACENUM).
- 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 Using traces and dumps in problem determination in 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 User exception trace entries in the *CICS Problem Determination Guide* for general information about user exception trace entry points; programming information is in Making trace entries in the *CICS Customization Guide*.

Monitoring application performance

CICS monitoring provides information about the performance of your application transactions. Use the MONITOR command for user event monitoring points.

In addition to the monitoring data collected from a system defined elsewhere, monitoring points (EMPs) in CICS, a user application program can contribute data to user fields in 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 16384 bytes of your own data in each performance monitoring record. In those 16384 bytes, you can have any combination of the following:

- In the range 0 - 256 counters
- In the range 0 - 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.

Dump

CICS dump allows you to specify areas of main storage to be dumped, by means of the DUMP TRANSACTION command, 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 PERFORM DUMP in the *CICS System Programming Reference* for programming information about this command.

You can format the contents of the dump data set and you can print them offline using the CICS dump utility program (DFHDU680) for transaction dumps or the interactive problem control system (IPCS) for system dumps. Instructions on using these programs are given in the Dump utility program 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.

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: lists of the CICS system initialization parameters and the FILE, PROGRAM, TRANSACTION and TERMINAL resource definitions in the CICS system. 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 programs, indexed by address, at the end of the printed dump.

The QUERY SECURITY command

You can use the **EXEC CICS QUERY SECURITY** command to determine whether a terminal user has access to resources that are defined to the external security manager.

You can use the **QUERY SECURITY** command with RACF® or any equivalent external security manager (ESM). The terminal user in this context is the user invoking the transaction that contains the **QUERY SECURITY** command.

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

This topic covers typical uses of the QUERY SECURITY command. One of which 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. RESID values in 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 SEC system initialization parameter in Reference -> System definition.

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 Security checking using the QUERY SECURITY command in the *CICS RACF Security Guide*.

CICS intercommunication

The areas to consider when you write applications that communicate with other CICS systems are summarized.

For further information about CICS intercommunication, see Getting started with intercommunication in Getting started.

You can run application programs in a CICS intercommunication environment using one or more of the following facilities:

Transaction routing
Transaction routing enables a terminal in one CICS system to run a transaction in another CICS system. See “Transaction routing” on page 370.

Function shipping
Function shipping enables your application program to access resources in another CICS system. See “Function shipping” on page 371.

Distributed program link (DPL)
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 371.

Asynchronous processing
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 383.

Distributed transaction processing (DTP)
DTP enables a CICS transaction to communicate with a transaction running in another system. Two interfaces are 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 383.

Common Programming Interface Communications (CPI-C)
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 384.

External CICS interface (EXCI)
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. CICS supports MVS resource recovery services (RRS) in applications that use the external CICS interface. See "External CICS interface (EXCI)" on page 384.

For details about the intercommunication aspects of the CICS Front End Programming Interface (FEPI), see Introduction to FEPI in Product overview.

Design considerations

If your application program uses more than one of the facilities listed earlier, you must allow for 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, you can use COBOL, C, C++, PL/I, or assembler language to write application programs that use CICS intercommunication facilities. However, for DTP application programs that hold APPC unmapped conversations using the EXEC CICS API, you can use only C, C++, or assembler language.

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. For more information about this utility, see CICS Interdependency Analyzer. For more information about transaction affinity, see "Affinity" on page 312.

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 cannot 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 unavailable (SYSIDERR), if a request is invalid (ISCINVREQ), or if the mirror transaction abends (AIPM abend for IPIC, 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 definition.

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. By specifying the remote system name in a XPCREQ global user exit

The basic flow in distributed program link is described in CICS distributed program link. The following terms, illustrated in Figure 113, 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.

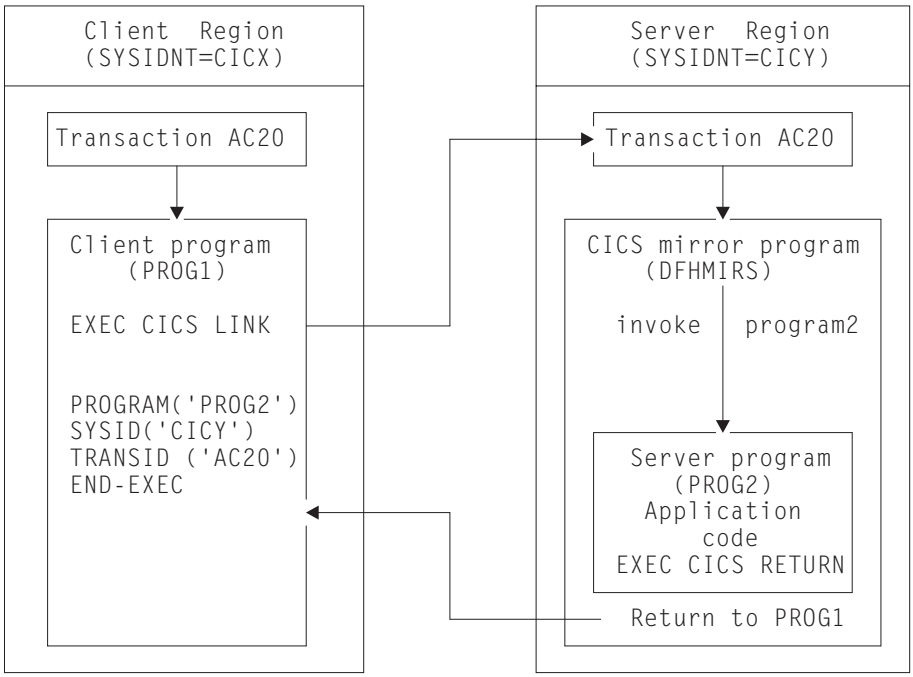


Figure 113. Illustration of distributed program link

Using the distributed program link function

Use these tables to understand the options on the distributed program link function.

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 28 on page 382.
- 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 26 are used on the LINK command, and the options shown in Table 27 on page 374 are used in the PROGRAM resource definition in support of the distributed program link facility.

Table 26. 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.
LENGTH	Specifies a halfword binary value that is the length in bytes of the COMMAREA. Specify a valid value (not zero) for the LENGTH option, because a length of zero when used on a DPL request can lead to unpredictable results and possibly a program failure.
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 sync point 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*, but note that for a distributed program link you cannot specify the INPUTMSG or INPUTMSGLEN options.

Table 27. 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 *CICS Intercommunication Guide*.
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 shows the syntax of the command.

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 114. COBOL example of a distributed program link

1. The program name of the server program.

   A program might have different names in the client and server regions. The name you specify on the LINK command depends on whether 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, or to any program name returned by the dynamic routing program. In this situation, 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:

a. The REMOTENAME attribute of the program definition
b. 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. You can use this option 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 302 has more information about using COMMAREAs to pass data to other programs.

   Ensure that you specify a valid value (not zero) for the LENGTH option, because a length of zero when used on a DPL request can lead to unpredictable results and possibly a program failure.

   When you use a COMMAREA, the program that is linked to must verify that the EIBCALEN field in the EIB of the task matches what the program expects. For more information, see "COMMAREA" on page 302.

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 as follows:

   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 lookup to obtain the netname (z/OS Communications Server 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 cannot 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 Enabling for specific invocation types in 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 115 on page 378 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 116 on page 378 shows an example of using distributed program link without the SYNCONRETURN option.

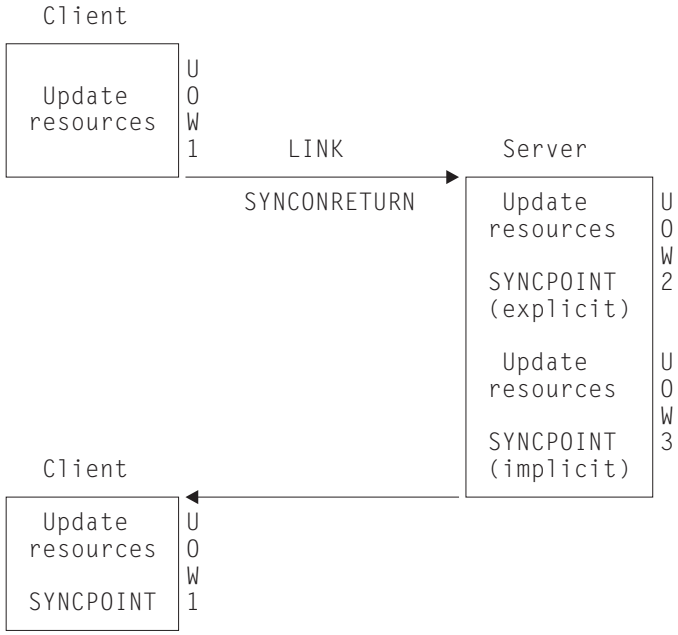


Figure 115. Using distributed program link with the SYNCONRETURN option

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.

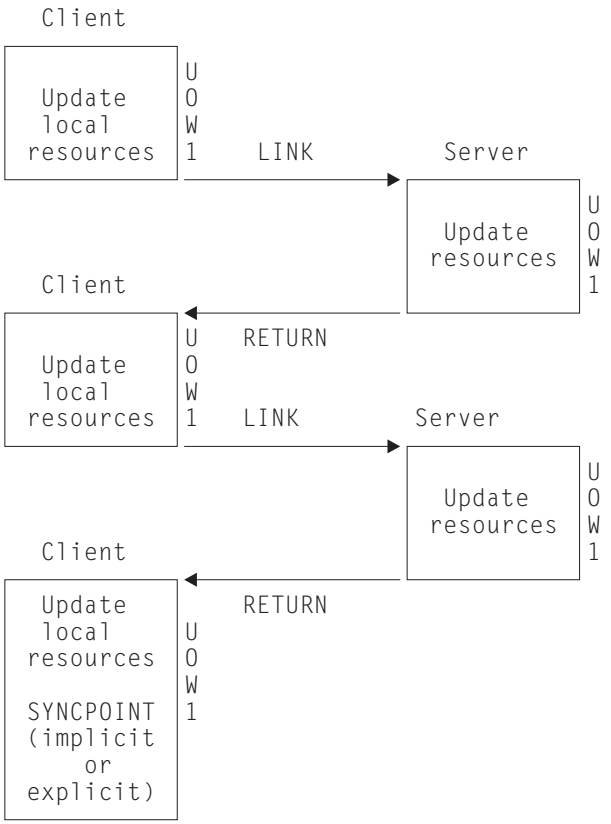


Figure 116. Using distributed program link without the SYNCONRETURN option

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

Consider the following factors 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 or IPCONN MIRRORLIFE set to UOW or TASK). 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 CICS function shipping in Getting started 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.

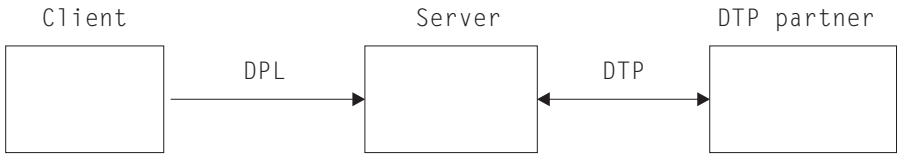


Figure 117. Example of mixing DPL and DTP

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 28 on page 382.

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.

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, before 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, 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 CICS API commands in Reference -> Application development 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 user ID specified on the **USERID** parameter (for preset security) of the SESSIONS resource definition, if present
2. The user ID specified on the **SECURITYNAME** parameter of the connection resource definition, if present and no preset security user ID is defined on the sessions
3. The user ID specified on the **DFLTUSER** system initialization parameter of the server region, if neither the sessions nor connection definitions specify a user ID

If any value other than LOCAL is specified for ATTACHSEC, the signed-on user ID is the one received in the attach request from the client region.

For more information, see Intercommunication security in Securing

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 which are returned to the client and server programs.

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 can also be raised. See CICS API commands in Reference -> Application development 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 failure of the mirror, and either:
   - Takes an implicit sync point by normal transaction termination
   - *or* Issues an explicit sync point request.
2. The remote mirror transaction performed no recoverable work within the scope of the unit of work of the application program. 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 an unrecoverable 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 28. 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 LINK in Reference -> Application development .

Table 28. 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

Table 28. API commands prohibited in programs invoked by DPL (continued)

Command	Options
START	TERMID, where its value is the ID of the intersystem session. (That is, where the issuing task's principal facility is a session rather than a terminal.)
START CHANNEL	TERMID, where its value is the ID of the intersystem session. (That is, where the issuing task's principal facility is a session rather than a terminal.)
SYNCPOINT	Can be issued in server region if SYNCONRETURN specified on LINK
SYNCPOINT ROLLBACK	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 two transactions can have exclusive control of a session and can “converse”. DTP is useful when you need remote resources to be processed remotely or if you need to transfer data between systems.

By using DTP, you can design 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 and applications that use the CICS intercommunication facilities.

DTP can be used with various partners, including both CICS and non-CICS platforms, as long as they support APPC. For further information about DTP, see Distributed transaction processing overview in Product overview and Getting started with intercommunication in Getting started.

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 of benefit.

Common Programming Interface Communications (CPI Communications) is the communication element of the Systems Applications Architecture (SAA) Common Programming Interface (CPI).

The CPI Communications interface can converse with applications on any system that provides an APPC API. This includes applications on CICS platforms. You can 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.

Calls from the application to the CPI Communications interface are resolved by link-editing the application with the CICS CPI Communications stub (DFHCPLC). For information about how to do this, see "Including the CICS-supplied interface modules" on page 135.

The CPI Communications API is defined as a general call interface. The interface is described in the *Common Programming Interface Communications Reference*.

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 with 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 Introduction to the external CICS interface in Developing applications.

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 can now become part of the MVS unit of recovery associated with the EXCI client program.
- The CICS server unit of work can 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 5. Data mappings

Creating the mappings to transform application data to XML and vice versa.

© Copyright IBM Corp. 1989, 2013

Chapter 19. Mapping and transforming application data and XML

You can write application programs to transform application binary data into XML and vice versa. CICS supports a number of high-level languages and provides an XML assistant to map how the data is transformed during runtime processing. CICS uses the same technology for mapping application data to XML in SOAP messages, as part of the web services support.

Before you begin

You must have Java installed to run the XML assistant and to optionally perform validation when CICS transforms the data or XML.

About this task

The advantage of using this approach to transform application data to and from XML is that CICS goes beyond the capabilities offered by an XML parser. CICS can interpret the XML and perform record-based conversions of the application data. Therefore, it is easier and faster for you to create applications that work with XML using this approach.

The CICS XML assistant is a supplied utility that helps you to create the required artifacts to transform application binary data to XML or transform XML to application binary data. The XML assistant can create the artifacts in a bundle directory or another specified location on z/OS UNIX.

Procedure

1. Create the mappings using the XML assistant.
2. Create the resources in CICS to make the mappings available.
3. Create or update an application program to use the **TRANSFORM** API command. The application must use a channel-based interface.
4. Run the application to test that the transformation works as you intended. You can turn on validation to check that CICS converts the data correctly.

What to do next

These steps are explained in more detail in the following topics.

The CICS XML assistant

The CICS XML assistant is a set of batch utilities that can help you to transform XML into high-level language structures and vice versa. The assistant supports rapid deployment of applications that perform XML processing with the minimum amount of programming effort.

Using the XML assistant for CICS reduces the amount of code that you must write to parse or construct XML; CICS transforms data between XML fragments and the data structure of an application program.

© Copyright IBM Corp. 1989, 2013

The XML assistant can create an XML schema from a simple language structure, or a language structure from an existing XML schema, and supports COBOL, C/C++, and PL/I. It also generates metadata that CICS uses at run time to automatically convert XML data to binary application data or vice versa; the metadata is defined in an XML binding and stored on z/OS UNIX. The schema for the XML binding is in `/usr/lpp/cicsts/cicsts51/schemas/xmltransform/` directory on z/OS UNIX.

The CICS XML assistant comprises two utility programs:

DFHLS2SC
: This utility generates an XML schema and binding from a language structure.

DFHSC2LS
: This utility generates an XML binding and language structure that you can use in your application programs. You can use either a WSDL document or an XML schema as input.

The JCL procedures to run both programs are in the *hlq*.XDFHINST library, where *hlq* is the high-level qualifier of your CICS installation.

Related tasks:

"Generating mappings from language structures" on page 455
To create XML from application data or vice versa, you create the mappings to describe how CICS will transform the data and XML at run time. You can start from any application data record; for example, you can start with a COMMAREA, VSAM file, temporary storage queue, or a DB2 record.

"Generating mappings from an XML schema" on page 458
To create application data from XML or vice versa that conforms to an existing XML schema, you create the mappings to describe how CICS will transform the data at run time. You can start from an XML schema or a WSDL document.

Related information:

Diagnosing deployment errors

DFHLS2SC: high-level language to XML schema conversion

The DFHLS2SC cataloged procedure generates an XML schema and an XML binding file from a high-level language structure. Use DFHLS2SC when you want to create a CICS program that can parse or create XML. The job control statements for DFHLS2SC, its symbolic parameters, its input parameters and their descriptions are listed.

Job control statements for DFHLS2SC

JOB Starts the job.

EXEC Specifies the procedure name (DFHLS2SC).

INPUT.SYSUT1 DD
: Specifies the input. The input parameters are usually specified in the input stream. However, you can define them in a data set or in a member of a partitioned data set.

Symbolic parameters

The following symbolic parameters are defined in DFHLS2SC:

JAVADIR=*path*
Specifies the name of the Java directory that is used by DFHLS2SC. The value of this parameter is appended to /usr/lpp/ giving a complete path name of /usr/lpp/*path*.

Typically, you do not specify this parameter. The default value is the value that was supplied to the CICS installation job (DFHISTAR) in the **JAVADIR** parameter.

PATHPREF=*prefix*
Specifies an optional prefix that extends the z/OS UNIX directory path used on other parameters. The default is the empty string.

Typically, you do not specify this parameter. The default value is the value that was supplied to the CICS installation job (DFHISTAR) in the **JAVADIR** parameter.

SERVICE=*value*
Use this parameter only when directed to do so by IBM support.

TMPDIR=*tmpdir*
Specifies the location of a directory in z/OS UNIX that DFHLS2SC uses as a temporary work space. The user ID under which the job runs must have read and write permission to this directory.

The default value is /tmp.

TMPFILE=*tmpprefix*
Specifies a prefix that DFHLS2SC uses to construct the names of the temporary workspace files.

The default value is SC2WS.

USSDIR=*path*
Specifies the name of the CICS TS directory in the z/OS UNIX file system. The value of this parameter is appended to /usr/lpp/cicsts/ giving a complete path name of /usr/lpp/cicsts/*path*.

Typically, you do not specify this parameter. The default value is the value that was supplied to the CICS installation job (DFHISTAR) in the **USSDIR** parameter.

The temporary work space

DFHLS2SC creates the following three temporary files at run time:

tmpdir/*tmpprefix*.in

tmpdir/*tmpprefix*.out

tmpdir/*tmpprefix*.err

where:

tmpdir is the value specified in the **TMPDIR** parameter.

tmpprefix is the value specified in the **TMPFILE** parameter.

The default names for the files (when **TMPDIR** and **TMPFILE** are not specified), are as follows:

/tmp/LS2SC.in

/tmp/LS2SC.out

/tmp/LS2SC.err

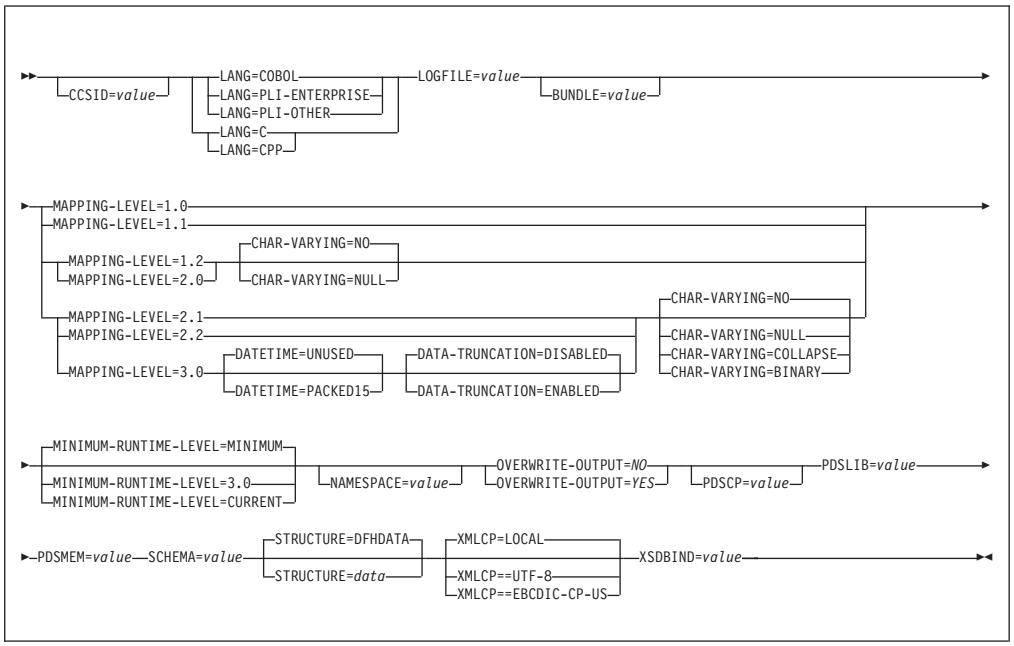
Important: DFHLS2SC does not lock access to the generated z/OS UNIX file names. Therefore, if two or more instances of DFHLS2SC run concurrently and use

the same temporary workspace files, nothing prevents one job from overwriting the workspace files while another job is using them, leading to unpredictable failures.

Therefore, you are advised to devise a naming convention and operating procedures that avoid this situation; for example, you can use the system symbolic parameter **SYSUID** to generate workspace file names that are unique to an individual user.

These temporary files are deleted before the end of the job.

Input parameters for DFHLS2SC



Parameter use

- You can specify the input parameters in any order.
- Each parameter must start on a new line.
- A parameter (and its continuation character, if you use one) must not extend beyond column 72; columns 73 to 80 must contain blanks.
- If a parameter is too long to fit on a single line, use an asterisk (*) character at the end of the line to indicate that the parameter continues on the next line. Everything (including spaces) before the asterisk is considered part of the parameter. For example:

  ```
  XSDBIND=xsdbinddir*
  /app1
  ```

  is equivalent to

  ```
  XSDBIND=xsdbinddir/app1
  ```
- A # character in the first character position of the line is a comment character. The line is ignored.
- A comma in the last character position of the line is an optional line separator, and is ignored.

Parameter descriptions

BUNDLE=*value*

Specifies the path and name of the bundle directory on z/OS UNIX. If you specify this value, the XML assistant generates a bundle containing the XSD binding. The path information on this parameter overrides any path information on the **XSDBIND** parameter.

You can optionally specify an archive file rather than a directory name. The XML assistant supports `.zip` and `.jar` archives. However, you must uncompress the archive before trying to install the BUNDLE resource.

If you do not specify this parameter, CICS places the XML schema and binding in the location specified on the **XSDBIND** parameter.

CCSID=*value*

Specifies the CCSID that is used at run time to encode character data in the application data structure. The value of this parameter overrides the value of the **LOCALCCSID** system initialization parameter. The *value* must be an EBCDIC CCSID that is supported by Java and z/OS conversion services. If you do not specify this parameter, the application data structure is encoded using the CCSID specified in the system initialization parameter.

You can use this parameter with any mapping level.

CHAR-VARYING={**NO**|**NULL**|**COLLAPSE**|**BINARY**}

Specifies how character fields in the language structure are mapped when the mapping level is 1.2 or higher. A character field in COBOL is a Picture clause of type X; for example, `PIC(X) 10`. A character field in C/C++ is a character array. This parameter does not apply to Enterprise and Other PL/I language structures. You can select these options:

NO Character fields are mapped to an `xsd:string` and are processed as fixed-length fields. The maximum length of the data is equal to the length of the field. NO is the default value for the **CHAR-VARYING** parameter for COBOL and PL/I at mapping levels 2.0 and earlier.

NULL Character fields are mapped to an `xsd:string` and are processed as null-terminated strings. CICS adds a terminating null character when transforming from an XML schema. The maximum length of the character string is calculated as one character less than the length indicated in the language structure. NULL is the default value for the **CHAR-VARYING** parameter for C/C++.

COLLAPSE
Character fields are mapped to an `xsd:string`. Trailing white space in the field is not included in the XML schema. COLLAPSE is the default *value* for the **CHAR-VARYING** parameter for COBOL and PL/I at mapping level 2.1 onwards.

BINARY
Character fields are mapped as an `xsd:base64binary` data type and are processed as fixed-length fields. The BINARY *value* on the **CHAR-VARYING** parameter is available only at mapping levels 2.1 and onwards.

DATA-TRUNCATION={**DISABLED**|**ENABLED**}

Specifies if variable length data is tolerated in a fixed length field structure:

DISABLED
If the data is less than the fixed length that CICS is expecting, CICS rejects the truncated data and issues an error message.

ENABLED
If the data is less than the fixed length that CICS is expecting, CICS tolerates the truncated data and processes the missing data as null values.

DATETIME={**UNUSED**|**PACKED15**}
Specifies if dateTime fields in the high-level language structure, including CICS ABSTIME values, are mapped as time stamps:

PACKED15
Any dateTime fields are mapped as time stamps.

UNUSED
Any dateTime fields are not mapped as time stamps. This mapping is the default.

You can set this parameter at a mapping level of 3.0.

LANG=COBOL
Specifies that the programming language of the high-level language structure is COBOL.

LANG=PLI-ENTERPRISE
Specifies that the programming language of the high-level language structure is Enterprise PL/I.

LANG=PLI-OTHER
Specifies that the programming language of the high-level language structure is a level of PL/I other than Enterprise PL/I.

LANG=C
Specifies that the programming language of the high-level language structure is C.

LANG=CPP
Specifies that the programming language of the high-level language structure is C++.

LOGFILE=*value*
The fully qualified z/OS UNIX name of the file into which DFHLS2SC writes its activity log and trace information. DFHLS2SC creates the file, but not the directory structure, if it does not already exist.

Typically, you do not use this file, but it might be requested by the IBM service organization if you encounter problems with DFHLS2SC.

MAPPING-LEVEL={1.0|1.1|1.2|2.0|2.1|2.2|3.0}
Specifies the level of mapping for the assistant to use when generating the XML binding and language structures. You are recommended to use the most recent mapping level that is available. If you are creating an XML binding for an Atom feed, you must use a mapping level of 3.0.

MINIMUM-RUNTIME-LEVEL={MINIMUM|3.0|CURRENT}
Specifies the minimum CICS runtime environment into which the XML binding can be deployed. If you select a level that does not match the other parameters that you have specified, you receive an error message. The options that you can select are as follows:

MINIMUM
The lowest possible runtime level of CICS is allocated automatically based on the parameters that you have specified.

3.0 Specify runtime level 3.0 or above if you want to use the CICS XML assistant and take advantage of advanced data mappings.

CURRENT
Use this runtime level to deploy the generated binding file into a CICS region that has the runtime environment set to the one used to generate the binding file.

NAMESPACE=*value*
Specifies the namespace for CICS to use in the generated XML schema. For Atom feeds, CICS provides this namespace in the Atom feed together with the Atom namespace.

If you do not specify this parameter, CICS generates a namespace automatically.

OVERWRITE-OUTPUT=*value*
Controls whether existing CICS BUNDLEs on the file system can be overwritten.

NO Any existing BUNDLE is not replaced. If an existing BUNDLE is found DFHLS2SC issues error message DFHPI9689E and terminates.

YES Any existing BUNDLE is replaced. If an existing BUNDLE is found then message DFHPI9683W is issued to inform you that the file has been replaced.

PDSCP=*value*
Specifies the code page that is used in the partitioned data set members, where *value* is a CCSID number or a Java code page number. If you do not specify this parameter, the z/OS UNIX System Services code page is used. For example, you might specify `PDSCP=037`.

PDSLIB=*value*
Specifies the name of the partitioned data set that contains the high-level language data structures to be processed.

Restriction: The records in the partitioned data set must have a fixed length of 80 bytes.

PDSMEM=*value*
Specifies the name of the partitioned data set member that contains the high-level language structures to be processed.

SCHEMA=*value*
The fully qualified z/OS UNIX name of the file into which the XML schema is written. The XML schema conforms to the WSDL 2.0 specification. DFHLS2SC creates the file, but not the directory structure, if it does not already exist.

STRUCTURE={**DFHDATA**|*data*}
The name of the top-level data structure in C and C++. The default is DFHDATA.

XMLCP={**LOCAL**|**UTF-8**|**EBCDIC-CP-US**}
Specifies the code page that is used to generate the XML binding.

LOCAL
This value is the default. It specifies that the XML is generated using the local code page and no encoding tag is generated in the XML schema.

UTF-8 Specifies that the XML is generated using the UTF-8 code page. An encoding tag is generated in the XML schema. If you specify this

option, you must ensure that the encoding remains correct when copying the XML schema between different platforms.

EBCDIC-CP-US
Specifies that the XML is generated using the US EBCDIC code page. An encoding tag is generated in the XML schema.

XSDBIND=*value*
The fully qualified z/OS UNIX name of the XSD binding. DFHLS2SC creates the file, but not the directory structure, if it does not already exist. The file extension is `.xsdbind`.

Related tasks:

"Generating mappings from language structures" on page 455
To create XML from application data or vice versa, you create the mappings to describe how CICS will transform the data and XML at run time. You can start from any application data record; for example, you can start with a COMMAREA, VSAM file, temporary storage queue, or a DB2 record.

DFHSC2LS: XML schema to high-level language conversion

The DFHSC2LS cataloged procedure generates a high-level language data structure and an XML binding from an XML schema or WSDL document. Use DFHSC2LS when you want to create a CICS program that can parse or create XML. The job control statements for DFHSC2LS, its symbolic parameters, its input parameters and their descriptions are listed.

Job control statements for DFHSC2LS

JOB Starts the job.

EXEC Specifies the procedure name (DFHSC2LS).

INPUT.SYSUT1 DD
Specifies the input. The input parameters are usually specified in the input stream. However, you can define them in a data set or in a member of a partitioned data set.

Symbolic parameters

The following symbolic parameters are defined in DFHSC2LS:

JAVADIR=*path*
Specifies the name of the Java directory that is used by DFHSC2LS. The value of this parameter is appended to `/usr/lpp/` giving a complete path name of `/usr/lpp/`*path*.

Typically, you do not specify this parameter. The default value is the value that was supplied to the CICS installation job (DFHISTAR) in the **JAVADIR** parameter.

PATHPREF=*prefix*
Specifies an optional prefix that extends the z/OS UNIX directory path used on other parameters. The default is the empty string.

Typically, you do not specify this parameter. The default value is the value that was supplied to the CICS installation job (DFHISTAR) in the **JAVADIR** parameter.

TMPDIR=*tmpdir*
Specifies the location of a directory in z/OS UNIX that DFHSC2LS uses as a temporary work space. The user ID under which the job runs must have read and write permission to this directory.

The default value is `/tmp`.

TMPFILE=*tmpprefix*
Specifies a prefix that DFHSC2LS uses to construct the names of the temporary workspace files.

The default value is `SC2LS`.

USSDIR=*path*
Specifies the name of the CICS TS directory in the z/OS UNIX file system. The value of this parameter is appended to `/usr/lpp/cicsts/` giving a complete path name of `/usr/lpp/cicsts/`*path*.

Typically, you do not specify this parameter. The default value is the value that was supplied to the CICS installation job (DFHISTAR) in the **USSDIR** parameter.

SERVICE=*value*
Use this parameter only when directed to do so by IBM support.

The temporary work space

DFHSC2LS creates the following three temporary files at run time:

tmpdir`/`*tmpprefix*`.in`

tmpdir`/`*tmpprefix*`.out`

tmpdir`/`*tmpprefix*`.err`

where:

tmpdir is the value specified in the **TMPDIR** parameter.

tmpprefix is the value specified in the **TMPFILE** parameter.

The default names for the files (when **TMPDIR** and **TMPFILE** are not specified), are as follows:

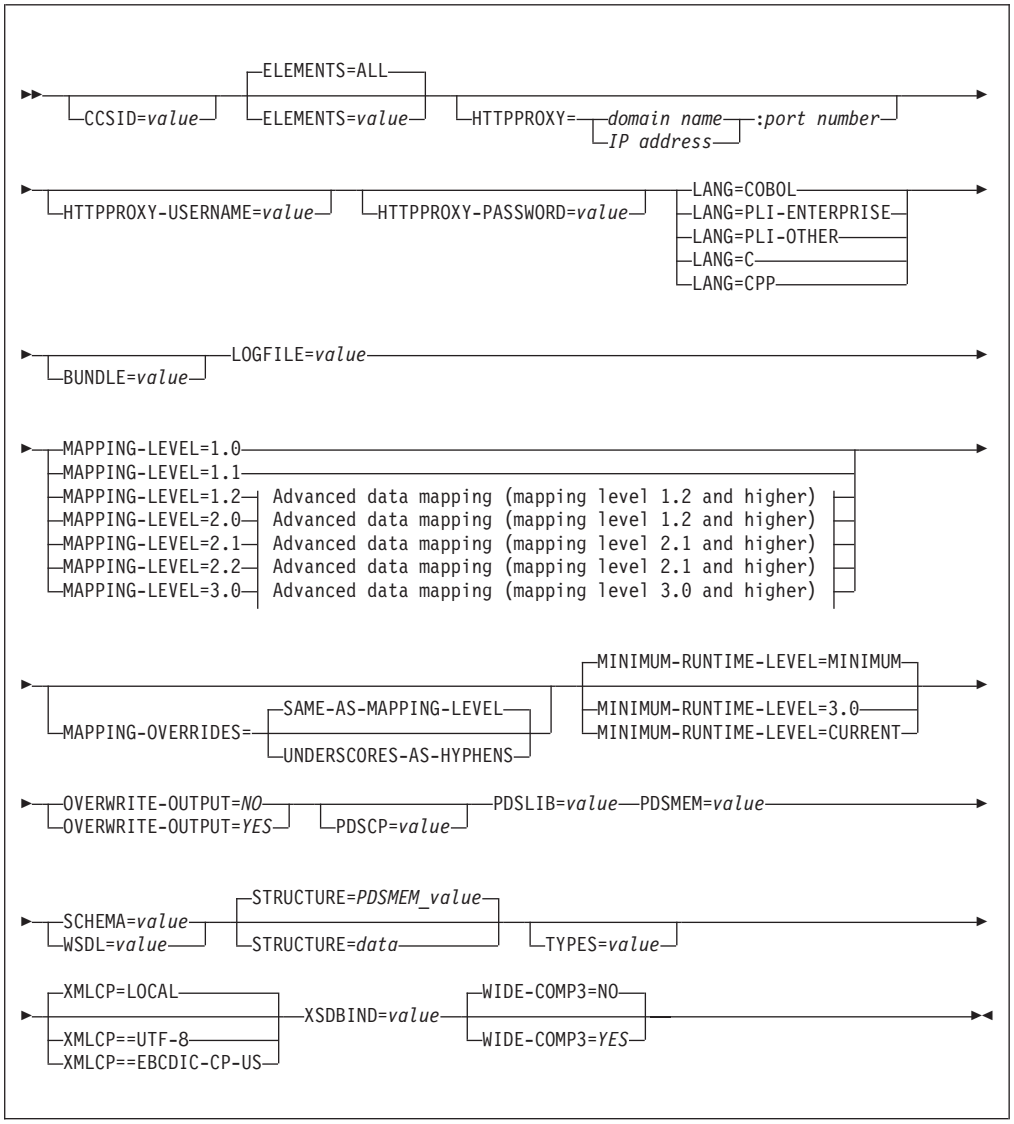
```
/tmp/SC2LS.in
/tmp/SC2LS.out
/tmp/SC2LS.err
```

Important: DFHSC2LS does not lock access to the generated z/OS UNIX file names. Therefore, if two or more instances of DFHSC2LS run concurrently, and use the same temporary workspace files, nothing prevents one job from overwriting the workspace files while another job is using them, leading to unpredictable failures.

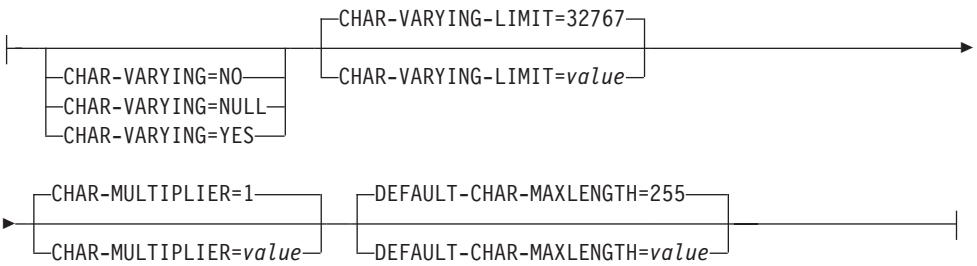
Therefore, you are advised to devise a naming convention and operating procedures that avoid this situation; for example, you can use the system symbolic parameter **SYSUID** to generate workspace file names that are unique to an individual user.

These temporary files are deleted before the end of the job.

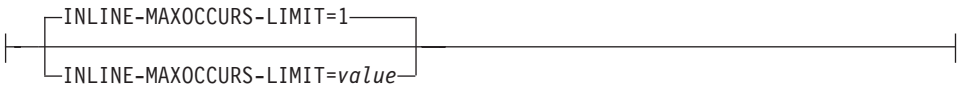
Input parameters for DFHSC2LS



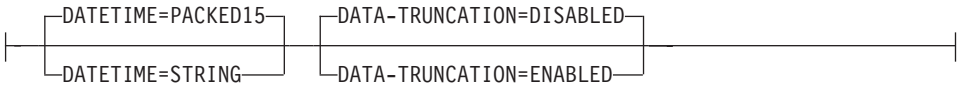
Advanced data mapping (mapping level 1.2 and higher):



Advanced data mapping (mapping level 2.1 and higher):



Advanced data mapping (mapping level 3.0):



Parameter use

- You can specify the input parameters in any order.
- Each parameter must start on a new line.
- A parameter (and its continuation character, if you use one) must not extend beyond column 72; columns 73 to 80 must contain blanks.
- If a parameter is too long to fit on a single line, use an asterisk (*) character at the end of the line to indicate that the parameter continues on the next line. Everything (including spaces) before the asterisk is considered part of the parameter. For example:

  ```
  XSDBIND=xsdbinddir*
  /app1
  ```

  is equivalent to

  ```
  XSDBIND=xsdbinddir/app1
  ```
- A # character in the first character position of the line is a comment character. The line is ignored.
- A comma in the last character position of the line is an optional line separator, and is ignored.

Parameter descriptions

BUNDLE=*value*

Specifies the path and name of the bundle directory on z/OS UNIX. If you specify this value, the XML assistant generates the XML binding in the bundle directory and creates a bundle manifest for you. The path information on this

parameter overrides any path information on the **XSDBIND** parameter. If you do not specify this parameter, CICS places the XML binding in the specified location.

CCSID=*value*
Specifies the CCSID that is used at run time to encode character data in the application data structure. The value of this parameter overrides the value of the **LOCALCCSID** system initialization parameter. The *value* must be an EBCDIC CCSID that is supported by Java and z/OS conversion services (see z/OS Unicode Services User's Guide and Reference). If you do not specify this parameter, the application data structure is encoded using the CCSID specified in the system initialization parameter.

You can use this parameter with any mapping level. However, if you want to deploy the generated files into a CICS TS 3.1 region, you must apply APAR PK23547 to achieve the minimum runtime level of code to install the web service binding file.

CHAR-MULTIPLIER={**1**|*value*}
Specifies the number of bytes to allow for each character when the mapping level is 1.2 or higher. The *value* of this parameter can be a positive integer in the range of 1 to 2 147 483 647. All nonnumeric character-based mappings are subject to this multiplier. Binary, numeric, zoned, and packed decimal fields are not subject to this multiplier.

This parameter can be useful if, for example, you are planning to use DBCS characters where you might opt for a multiplier of 3 to allow space for potential shift-out and shift-in characters around every double-byte character at run time.

CHAR-VARYING={**NO**|**NULL**|**YES**}
Specifies how variable-length character data is mapped when the mapping level is 1.2 or higher. Variable-length binary data types are always mapped to either a container or a varying structure. If you do not specify this parameter, the default mapping depends on the language specified. You can select these options:

NO Variable-length character data is mapped as fixed-length strings.

NULL Variable-length character data is mapped to null-terminated strings.

YES Variable-length character data is mapped to a CHAR VARYING data type in PL/I. In the COBOL, C, and C++ languages, variable-length character data is mapped to an equivalent representation that consists of two related elements: the data-length and the data.

CHAR-VARYING-LIMIT={**32767**|*value*}
Specifies the maximum size of binary data and variable-length character data that is mapped to the language structure when the mapping level is 1.2 or higher. If the character or binary data is larger than the value specified in this parameter, it is mapped to a container and the container name is used in the generated language structure. The *value* can range from 0 to the default 32 767 bytes.

DATA-TRUNCATION={**DISABLED**|**ENABLED**}
Specifies if variable length data is tolerated in a fixed length field structure:

DISABLED
If the data is less than the fixed length that CICS is expecting, CICS rejects the truncated data and issues an error message.

ENABLED
If the data is less than the fixed length that CICS is expecting, CICS tolerates the truncated data and processes the missing data as null values.

DATETIME={**PACKED15**|**STRING**}
Specifies that xsd:dateTime fields are mapped to CICS ABSTIME data format or to text:

PACKED15
xsd:dateTime fields are mapped to CICS ABSTIME format.

STRING
xsd:dateTime fields are mapped to text. This mapping is the same as all previous mappings levels.

You can use this parameter at a mapping level of 3.0.

DEFAULT-CHAR-MAXLENGTH={**255**|*value*}
Specifies the default array length of character data in characters for mappings where no length is implied in the XML schema document or WSDL document, when the mapping level is 1.2 or higher. The *value* of this parameter can be a positive integer in the range of 1 to 2 147 483 647.

ELEMENTS={**ALL**|*value*}
Defines a list of global element local names to enable. The default value of ALL indicates that all global elements are enabled.

HTTPPROXY={*domain name*|*IP address*}:*port number*
If your XML schema or WSDL document contains references to other XML schema or WSDL files that are located on the Internet, and the system on which you are running DFHSC2LS uses a proxy server to access the Internet, specify either the domain name or IP address and the port number of the proxy server. For example:

```
HTTPPROXY=proxy.example.com:8080
```

In other cases, this parameter is not required.

HTTPPROXY-USERNAME=*value*
Specifies the HTTP proxy user name to be used with **HTTPPROXY-PASSWORD** if the system on which you are running DFHSC2LS uses an HTTP proxy server to access the Internet, and the HTTP proxy server uses basic authentication. You can use this parameter only when you also specify **HTTPPROXY**.

HTTPPROXY-PASSWORD=*value*
Specifies the HTTP proxy password to be used with **HTTPPROXY-USERNAME** if the system on which you are running DFHSC2LS uses an HTTP proxy server to access the Internet, and the HTTP proxy server uses basic authentication. You can use this parameter only when you also specify **HTTPPROXY**.

INLINE-MAXOCCURS-LIMIT={**1**|*value*}
Specifies whether inline variable repeating content is used based on the `maxOccurs` attribute of the XML attribute.

The **INLINE-MAXOCCURS-LIMIT** parameter is available only at mapping level 2.1 onwards. The *value* of **INLINE-MAXOCCURS-LIMIT** can be a positive integer in the range of 0 to 32 767. A value of 0 indicates that inline mapping is not used. A value of 1 ensures that optional elements are mapped inline. If the *value* of the `maxOccurs` attribute is greater than the *value* of **INLINE-MAXOCCURS-LIMIT**, container based mapping is used; otherwise, inline mapping is used.

When deciding if you want variably repeating lists to be mapped inline, consider the length of a single item of recurring data. If you have few instances of long length, container-based mapping is preferable; if you have many instances of short length, inline mapping is likely to be preferable.

LANG=COBOL
Specifies that the programming language of the high-level language structure is COBOL.

LANG=PLI-ENTERPRISE
Specifies that the programming language of the high-level language structure is Enterprise PL/I.

LANG=PLI-OTHER
Specifies that the programming language of the high-level language structure is a level of PL/I other than Enterprise PL/I.

LANG=C
Specifies that the programming language of the high-level language structure is C.

LANG=CPP
Specifies that the programming language of the high-level language structure is C++.

LOGFILE=*value*
The fully qualified z/OS UNIX name of the file into which DFHSC2LS writes its activity log and trace information. DFHSC2LS creates the file, but not the directory structure, if it does not already exist.

Typically you do not use this file, but it might be requested by the IBM service organization if you encounter problems with DFHSC2LS.

MAPPING-LEVEL={1.0|1.1|1.2|2.0|2.1|2.2|3.0}
Specifies the level of mapping for the assistant to use when generating the XML binding and language structures. You are recommended to use the most recent mapping level that is available; for DFHSC2LS, you are recommended to use a mapping level of 3.0 or higher.

3.0 The `xsd:dateTime` data type maps to the CICS ASKTIME format.

MAPPING-OVERRIDES={SAME-AS-MAPPING-LEVEL|UNDERSCORES-AS-HYPHENS}
Specifies whether the default mapping behavior is overridden for the specified mapping level. The default value is `SAME-AS-MAPPING-LEVEL`.

UNDERSCORES-AS-HYPHENS
For COBOL only. Converts any underscores in the XML document to hyphens instead of the character X. This option improves the readability of the generated COBOL language structures. If any field name clashes occur, the fields are numbered to ensure they are unique. For more information, see "XML schema to COBOL mapping" on page 415.

MINIMUM-RUNTIME-LEVEL={MINIMUM|3.0|CURRENT}
Specifies the minimum CICS runtime environment into which the XML binding can be deployed. If you select a level that does not match the other parameters that you have specified, you receive an error message. The options that you can select are as follows:

MINIMUM
The lowest possible runtime level of CICS is allocated automatically based on the parameters that you have specified.

3.0 Specify runtime level 3.0 or above if you want to use the CICS XML assistant and take advantage of advanced data mappings.

CURRENT
Use this runtime level to deploy the generated XML binding into a CICS region that has the same runtime environment as the region used to generate the XML binding.

OVERWRITE-OUTPUT=*value*
Controls whether existing CICS BUNDLEs on the file system can be overwritten.

NO Any existing BUNDLE is not replaced. If an existing BUNDLE is found DFHLS2SC issues error message DFHPI9689E and terminates.

YES Any existing BUNDLE is replaced. If an existing BUNDLE is found then message DFHPI9683W is issued to inform you that the file has been replaced.

PDSCP=*value*
Specifies the code page that is used in the partitioned data set members, where *value* is a CCSID number or a Java code page number. If you do not specify this parameter, the z/OS UNIX System Services code page is used. For example, you might specify `PDSCP=037`.

PDSLIB=*value*
Specifies the name of the partitioned data set that contains the generated high-level language.

PDSMEM=*value*
Specifies the 1-6 character prefix that DFHSC2LS uses to generate the name of the partitioned data set member that will contain the high-level language structures.

DFHSC2LS generates a partitioned data set member for each operation. It generates the member name by appending a 2-digit number to the prefix.

SCHEMA=*value*
The fully qualified z/OS UNIX name of the file from which the XML schema is read. DFHSC2LS creates the file, but not the directory structure, if it does not already exist.

You can use either an XML schema or a WSDL document as input to DFHSC2LS. You must specify either this parameter or the **WSDL** parameter, but not both, to indicate from where the input comes.

STRUCTURE={*PDSMEM_value*|*data*}
The name of the top-level data structure in C and C++. The default value is the value of the **PDSMEM** parameter.

TYPES=*value*
Defines a list of global type local names to enable. A *value* of ALL indicates that all global types are enabled. By default, global types are not enabled.

WIDE-COMP=NO|YES
For COBOL only. Controls the maximum size of the packed decimal variable length in the COBOL language structure.

NO DFHWS2LS limits the packed decimal variable length to 18 when generating the COBOL language structure type COMP-3. If the packed decimal size is greater than 18, message DFHPI9022W is issued to indicate that the specified type is being restricted to a total of 18 digits.

YES DFHWS2LS supports the maximum size of 31 when generating the COBOL language structure type COMP-3.

WSDL=*value*
The fully qualified z/OS UNIX name of the WSDL document.

You can use either an XML schema or a WSDL document as input to DFHSC2LS. You must specify either this parameter or the **SCHEMA** parameter, but not both, to indicate from where the input comes.

XMLCP={**LOCAL**|**UTF-8**|**EBCDIC-CP-US**}
Specifies the code page that is used to generate the XML binding.

LOCAL
This value is the default. It specifies that the XML is generated using the local code page and no encoding tag is generated in the XML schema.

UTF-8 Specifies that the XML is generated using the UTF-8 code page. An encoding tag is generated in the XML schema. If you specify this option, you must ensure that the encoding remains correct when copying the XML schema between different platforms.

EBCDIC-CP-US
Specifies that the XML is generated using the US EBCDIC code page. An encoding tag is generated in the XML schema.

XSDBIND=*value*
The fully qualified z/OS UNIX name of the XML binding. DFHSC2LS creates the file, but not the directory structure, if it does not already exist. The file extension is `.xsdbind`.

Mapping levels for the CICS assistants

A mapping is the set of rules that specifies how information is converted between language structures and XML schemas. To benefit from the most sophisticated mappings available, you are recommended to set the **MAPPING-LEVEL** parameter in the CICS assistants to the latest level.

Each level of mapping inherits the function of the previous mapping, with the highest level of mapping offering the best capabilities available. The highest mapping level provides more control over data conversion at run time and removes restrictions on support for certain data types and XML elements.

You can set the **MAPPING-LEVEL** parameter to an earlier level if you want to redeploy applications that were previously enabled at that level.

Mapping level 3.0

Mapping level 3.0 is compatible with a CICS TS 4.1 region.

This mapping level provides the following support:

- DFHSC2LS and DFHWS2LS map `xsd:dateTime` data types to CICS ASKTIME format.
- DFHLS2WS can generate a WSDL document and web service binding from an application that uses many containers rather than just one container.
- Tolerating truncated data that is described by a fixed length data structure. You can set this behavior by using the **DATA-TRUNCATION** parameter on the CICS assistants.

Mapping level 2.2 and higher

Mapping level 2.2 is compatible with a CICS TS 3.2 region, with APAR PK69738 applied, and higher.

At mapping level 2.2 and higher, DFHSC2LS and DFHWS2LS support the following XML mappings:

- Fixed values for elements
- Substitution groups
- Abstract data types
- XML schema <sequence> elements can nest inside <choice> elements

DFHSC2LS and DFHWS2LS provide enhanced support for the following XML mappings:

- Abstract elements
- XML schema <choice> elements

Mapping level 2.1 and higher

Mapping level 2.1 is compatible with a CICS TS 3.2 region, with APAR PK59794 applied, and higher.

This mapping level includes greater control over the way variable content is handled with the new **INLINE-MAXOCCURS-LIMIT** parameter and new values on the **CHAR-VARYING** parameter.

At mapping level 2.1 and higher, DFHSC2LS and DFHWS2LS offer the following new and improved support for XML mappings:

- The XML schema <any> element
- The `xsd:anyType` type
- Toleration of abstract elements
- The **INLINE-MAXOCCURS-LIMIT** parameter
- The `minOccurs` attribute

The **INLINE-MAXOCCURS-LIMIT** parameter specifies whether variably repeating lists are mapped inline. For more information on mapping variably repeating content inline, see Variable arrays of elements.

Support for the `minOccurs` attribute has been enhanced on the XML schema <sequence>, <choice>, and <all> elements. If `minOccurs="0"`, the CICS assistant treats these element as though the `minOccurs="0"` attribute is also an attribute of all its child elements.

At mapping level 2.1 and higher, DFHLS2SC and DFHLS2WS support the following XML mappings:

- FILLER fields in COBOL and PL/I are ignored
- A value of COLLAPSE for the **CHAR-VARYING** parameter
- A value of BINARY for the **CHAR-VARYING** parameter

FILLER fields in COBOL and PL/I are ignored; they do not appear in the generated XML schema and an appropriate gap is left in the data structures at run time.

COLLAPSE causes CICS to ignore trailing spaces in text fields.

BINARY provides support for binary fields. This value is useful when converting COBOL into an XML schema. This option is available only on SBCS character arrays and allows the array to be mapped to fixed-length xsd:base64Binary fields rather than to xsd:string fields.

Mapping level 1.2 and higher

Mapping level 1.2 is compatible with a CICS TS 3.1 region and higher.

Greater control is available over the way character and binary data are transformed at run time with these additional parameters on the batch tools:

- **CHAR-VARYING**
- **CHAR-VARYING-LIMIT**
- **CHAR-MULTIPLIER**
- **DEFAULT-CHAR-MAXLENGTH**

If you decide to use the **CHAR-MULTIPLIER** parameter in DFHSC2LS or DFHWS2LS, note that the following rules apply after the value of this parameter is used to calculate the amount of space required for character data.

- DFHSC2LS and DFHWS2LS provide these mappings:
  - Variable-length character data types that have a maximum length of more than 32 767 bytes map to a container. You can use the **CHAR-VARYING-LIMIT** parameter to set a lower limit. A 16-byte field is created in the language structure to store the name of the container. At run time, the character data is stored in a container and the container name is put in the language structure.
  - Variable-length character data types that have a maximum length of less than 32 768 bytes map to a VARYING structure for all languages except C/C++ and Enterprise PL/I. In C/C++, these data types are mapped to null-terminated strings, and in Enterprise PL/I these data types are mapped to VARYINGZ structures. You can use the **CHAR-VARYING** parameter to select the way that variable-length character data is mapped.
  - Variable-length binary data that has a maximum length of less than 32 768 bytes maps to a VARYING structure for all languages. If the maximum length is equal to or greater than 32 768 bytes, the data is mapped to a container. A 16-byte field is created in the language structure to store the name of the container. At run time, the binary data is stored in a container and the container name is put in the language structure.

If you have character data types in the XML schema that do not have a length associated with them, you can assign a default length using the **DEFAULT-CHAR-MAXLENGTH** parameter in DFHWS2LS or DFHSC2LS.

DFHLS2SC and DFHLS2WS provide these mappings:

- Character fields map to an xsd:string data type and can be processed as fixed-length fields or null-terminated strings at run time. You can use the **CHAR-VARYING** parameter to select the way that variable-length character data is handled at run time for all languages except PL/I.
- Base64Binary data types map to a container if the maximum length of the data is greater than 32 767 bytes or when the length is not defined. If the length of the data is 32 767 or less, the base64Binary data type is mapped to a VARYING structure for all languages.

Mapping level 1.1 and higher

Mapping level 1.1 is compatible with a CICS TS 3.1 region and higher.

This mapping level provides improved mapping of XML character and binary data types, in particular when mapping data of variable length that has `maxLength` and `minLength` attributes defined with different values in the XML schema. Data is handled in the following ways:

- Character and binary data types that have a fixed length that is greater than 16 MB map to a container for all languages except PL/I. In PL/I, fixed-length character and binary data types that are greater than 32 767 bytes are mapped to a container. A 16-byte field is created in the language structure to store the name of the container. At run time, the fixed-length data is stored in a container and the container name is put in the language structure.

  Because containers are variable in length, fixed-length data that is mapped to a container is not padded with spaces or nulls, or truncated, to match the fixed length specified in the XML schema or web service description. If the length of the data is significant, you can either write your application to check it or turn validation on in the CICS region. Both SOAP and XML validation have a significant performance impact.
- XML schema <list> and <union> data types map to character fields.
- Schema-defined XML attributes are mapped rather than ignored. A maximum of 255 attributes is allowed for each XML element. See Support for XML attributes for further information.
- The `xsi:nil` attribute is supported. See Support for XML attributes for further information.

Mapping level 1.1 only

Mapping level 1.1 is compatible with a CICS TS 3.1 region and higher.

This mapping level provides improved mapping of XML character and binary data types, in particular when mapping data of variable length that has `maxLength` and `minLength` attributes defined with different values in the XML schema. Data is handled in the following ways:

- Variable-length binary data types map to a container. A 16-byte field is created in the language structure to store the name of the container. At run time, the binary data is stored in a container and the container name is put in the language structure.
- Variable-length character data types that have a maximum length greater than 32 767 bytes map to a container. A 16-byte field is created in the language structure to store the name of the container. At run time, the character data is stored in a container and the container name is put in the language structure.
- Character and binary data types that have a fixed length of less than 16 MB map to fixed-length fields for all languages except PL/I. In PL/I, fixed-length character and binary data types that are 32 767 bytes or less map to fixed-length fields.
- CICS encodes and decodes data in the hexBinary format but not in base64Binary format. Base64Binary data types in the XML schema map to a field in the language structure. The size of the field is calculated using the formula: $4\times(\text{ceil}(z/3))$ where:
  - z is the length of the data type in the XML schema.
  - ceil(x) is the smallest integer greater than or equal to x.

If the length of *z* is greater than 24 566 bytes, the resulting language structure fails to compile. If you have base64Binary data that is greater than 24 566 bytes, you are recommended to use a mapping level of 1.2. With mapping level 1.2, you can map the base64Binary data to a container instead of using a field in the language structure.

Mapping level 1.0 only

Mapping level 1.0 is compatible with a CICS TS 3.1 region and higher.

Note the following limitations, which have been modified in later mapping levels:

- DFHSC2LS and DFHWS2LS map character and binary data types in the XML schema to fixed-length fields in the language structure. Look at this partial XML schema:

```
<xsd:element name="example">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:maxLength value="33000"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

  That partial XML schema appears in a COBOL language structure like this:

```
15 example     PIC X(33000)
```

- CICS encodes and decodes data in the hexBinary format but not in base64Binary format. DFHSC2LS and DFHWS2LS map Base64Binary data to a fixed-length character field, the contents of which must be encoded or decoded by the application program.
- DFHSC2LS and DFHWS2LS ignore XML attributes during processing.
- DFHLS2SC and DFHLS2WS interpret character and binary fields in the language structure as fixed-length fields and map those fields to XML elements that have a `maxLength` attribute. At run time, the fields in the language structure are filled with spaces or nulls if insufficient data is available.

Related reference:

"Support for XML attributes" on page 439
XML schemas can specify attributes that are allowed or required in XML. The CICS assistant utilities DFHWS2LS and DFHSC2LS ignore XML attributes by default. To process XML attributes that are defined in the XML schema, the value of the **MAPPING-LEVEL** parameter must be 1.1 or higher.

"Data mapping limitations when using the CICS assistants" on page 409
CICS supports bidirectional data mappings between high-level language structures and XML schemas or WSDL documents that conform to WSDL version 1.1 or 2.0, with certain limitations. These limitations apply only to the DFHWS2LS and DFHSC2LS tools and vary according to the mapping level.

High-level language and XML schema mapping

Use the CICS assistants to generate mappings between high-level language structures and XML schemas or WSDL documents. The CICS assistants also generate XML schemas or WSDL documents from high-level language data structures, or vice-versa.

Utility programs DFHSC2LS and DFHLS2SC are collectively known as the CICS XML assistant. Utility programs DFHWS2LS and DFHLS2WS are collectively known as the CICS Web services assistant.

- DFHLS2SC and DFHLS2WS map high-level language structures to XML schemas and WSDL documents respectively.
- DFHSC2LS and DFHWS2LS map XML schemas and WSDL documents to high-level language structures.

The two mappings are not symmetrical:

- If you process a language data structure with DFHLS2SC or DFHLS2WS and then process the resulting XML schema or WSDL document with the complementary utility program (DFHSC2LS or DFHWS2LS respectively), do not expect the final data structure to be the same as the original. However, the final data structure is logically equivalent to the original.
- If you process an XML schema or WSDL document with DFHSC2LS or DFHWS2LS and then process the resulting language structure with the complementary utility program (DFHLS2SC or DFHLS2WS respectively), do not expect the XML schema in the final XML schema or WSDL document to be the same as the original.
- In some cases, DFHSC2LS and DFHWS2LS generate language structures that are not supported by DFHLS2SC and DFHLS2WS.

You must code language structures processed by DFHLS2SC and DFHLS2WS according to the rules of the language, as implemented in the language compilers that CICS supports.

Related tasks:

"Generating mappings from language structures" on page 455
To create XML from application data or vice versa, you create the mappings to describe how CICS will transform the data and XML at run time. You can start from any application data record; for example, you can start with a COMMAREA, VSAM file, temporary storage queue, or a DB2 record.

Data mapping limitations when using the CICS assistants

CICS supports bidirectional data mappings between high-level language structures and XML schemas or WSDL documents that conform to WSDL version 1.1 or 2.0, with certain limitations. These limitations apply only to the DFHWS2LS and DFHSC2LS tools and vary according to the mapping level.

Limitations at all mapping levels

- Only SOAP bindings that use literal encoding are supported. Therefore, you must set the `use` attribute to a value of `literal`; `use="encoded"` is not supported.
- Data type definitions must be encoded using the XML Schema Definition language (XSD). In the schema, data types used in the SOAP message must be explicitly declared.
- The length of some keywords in the Web services description is limited. For example, operation, binding, and part names are limited to 255 characters. In some cases, the maximum operation name length might be slightly shorter.
- Any SOAP faults defined in the Web service description are ignored. If you want a service provider application to send a SOAP fault message, use the **EXEC CICS SOAPFAULT** command.
- DFHWS2LS and DFHSC2LS support only a single <xsd:any> element in a particular scope. For example, the following schema fragment is not supported:

```
<xsd:sequence>
 <xsd:any/>
 <xsd:any/>
</xsd:sequence>
```

Here, <xsd:any> can specify minOccurs and maxOccurs if required. For example, the following schema fragment is supported:

```
<xsd:sequence>
 <xsd:any minOccurs="2" maxOccurs="2"/>
</xsd:sequence>
```

- Cyclic references are not supported. For example, where type A contains type B which, in turn, contains type A.
- Recurrence is not supported in group elements, such as <xsd:choice>, <xsd:sequence>, <xsd:group>, or <xsd:all> elements. For example, the following schema fragment is not supported:

  ```
  <xsd:choice maxOccurs="2">
     <xsd:element name="name1" type="string"/>
  </xsd:choice>
  ```

  The exception is at mapping level 2.1 and higher, where maxOccurs="1" and minOccurs="0" are supported on these elements.
- DFHSC2LS and DFHWS2LS do not support data types and elements in the SOAP message that are derived from the declared data types and elements in the XML schema either from the xsi:type attribute or from a substitution group, except at mapping level 2.2 and higher if the parent element or type is defined as abstract.
- Embedded <xsd:sequence> and <xsd:group> elements inside an <xsd:choice> element are not supported prior to mapping level 2.2. Embedded <xsd:choice> and <xsd:all> elements inside an <xsd:choice> element are never supported.

Improved support at mapping level 1.1 and higher

When the mapping level is 1.1 or higher, DFHWS2LS provides support for the following XML elements and element type:

- The <xsd:list> element.
- The <xsd:union> element.
- The xsd:anySimpleType type.
- The <xsd:attribute> element. At mapping level 1.0 this element is ignored.

Improved support at mapping level 2.1 and higher

When the mapping level is 2.1 or higher, DFHWS2LS supports the following XML elements and element attributes:

- The <xsd:any> element.
- The xsd:anyType type.
- Abstract elements. In earlier mapping levels, abstract elements are supported only as nonterminal types in an inheritance hierarchy.
- The maxOccurs and minOccurs attributes on the <xsd:all>, <xsd:choice>, and <xsd:sequence> elements, only when maxOccurs="1" and minOccurs="0".
- "FILLER" fields in COBOL and "*" fields in PL/I are suppressed. The fields do not appear in the generated WSDL and an appropriate gap is left in the data structures at run time.

Improved support at mapping level 2.2 and higher

When the mapping level is 2.2 or higher, DFHSC2LS and DFHWS2LS provide improved support for the <xsd:choice> element, supporting a maximum of 255

options in the <xsd:choice> element. For more information on <xsd:choice> support, see “Support for <xsd:choice>” on page 444.

At mapping level 2.2 and higher, the CICS assistants support the following XML mappings:

- Substitution groups
- Fixed values for elements
- Abstract data types

Embedded <xsd:sequence> and <xsd:group> elements inside an <xsd:choice> element are supported at mapping level 2.2 and higher. For example, the following schema fragment is supported:

```
<xsd:choice>
   <xsd:element name="name1" type="string"/>
   <xsd:sequence/>
</xsd:choice>
```

If the parent element or type in the SOAP message is defined as abstract, DFHSC2LS and DFHWS2LS support data types and elements that are derived from the declared data types and elements in the XML schema.

Improved support at mapping level 3.0 and higher

When the mapping level is 3.0 or higher, the CICS assistants support the following mapping improvements:

- DFHSC2LS and DFHWS2LS map `xsd:dateTime` data types to CICS ASKTIME format.
- DFHLS2WS can generate a WSDL document and web service binding from an application that uses many containers rather than just one container.
- Tolerating truncated data that is described by a fixed length data structure. You can set this behavior by using the **DATA-TRUNCATION** parameter on the CICS assistants.

Related reference:

“Mapping levels for the CICS assistants” on page 404
A mapping is the set of rules that specifies how information is converted between language structures and XML schemas. To benefit from the most sophisticated mappings available, you are recommended to set the **MAPPING-LEVEL** parameter in the CICS assistants to the latest level.

COBOL to XML schema mapping

The DFHLS2SC and DFHLS2WS utility programs support mappings between COBOL data structures and XML schema definitions.

COBOL names are converted to XML names according to the following rules:

1. Duplicate names are made unique by the addition of one or more numeric digits.

   For example, two instances of `year` become `year` and `year1`.
2. Hyphens are replaced by underscore characters. Strings of contiguous hyphens are replaced by contiguous underscores.

   For example, `current-user--id` becomes `current_user__id`.
3. Segments of names that are delimited by hyphens and that contain only uppercase characters are converted to lowercase.

   For example, `CA-REQUEST-ID` becomes `ca_request_id`.

4. A leading underscore character is added to names that start with a numeric character.

   For example, `9A-REQUEST-ID` becomes `_9a_request_id`.

CICS maps COBOL data description elements to schema elements according to the following table. COBOL data description elements that are not shown in the table are not supported by DFHLS2SC or DFHLS2WS. The following restrictions also apply:

- Data description items with level numbers of 66 and 77 are not supported. Data description items with a level number of 88 are ignored.
- The following clauses on data description entries are not supported:

  OCCURS DEPENDING ON

  OCCURS INDEXED BY

  REDEFINES

  RENAMES; that is level 66

  DATE FORMAT
- The following clauses on data description items are ignored:

  BLANK WHEN ZERO

  JUSTIFIED

  VALUE
- The SIGN clause SIGN TRAILING is supported. The SIGN clause SIGN LEADING is supported only when the mapping level specified in DFHLS2SC or DFHLS2WS is 1.2 or higher.
- SEPARATE CHARACTER is supported at a mapping level of 1.2 or higher for both SIGN TRAILING and SIGN LEADING clauses.
- The following phrases on the USAGE clause are not supported:

  OBJECT REFERENCE

  POINTER

  FUNCTION-POINTER

  PROCEDURE-POINTER
- The following phrases on the USAGE clause are supported at a mapping level of 1.2 or higher:

  COMPUTATIONAL-1

  COMPUTATIONAL-2
- The only PICTURE characters supported for DISPLAY and COMPUTATIONAL-5 data description items are 9, S, and Z.
- The PICTURE characters supported for PACKED-DECIMAL data description items are 9, S, V, and Z.
- The only PICTURE characters supported for edited numeric data description items are 9 and Z.
- If the **MAPPING-LEVEL** parameter is set to 1.2 or higher and the **CHAR-VARYING** parameter is set to NULL, character arrays are mapped to an `xsd:string` and are processed as null-terminated strings.
- If the **MAPPING-LEVEL** parameter is set to 1.2 or higher and the **CHAR-VARYING** parameter is set to BINARY, character arrays are mapped to `xsd:base64Binary` and are processed as binary data.
- If the **MAPPING-LEVEL** parameter is set to 1.2 or higher and the **CHAR-VARYING** parameter is set to COLLAPSE, trailing white space is ignored for strings.

COBOL data description	Schema simpleType
PIC X(*n*) PIC A(*n*) PIC G(*n*) DISPLAY-1 PIC N(*n*)	<xsd:simpleType> <xsd:restriction base="xsd:string"> <xsd:maxlength value="*n*"/> <xsd:whiteSpace value="preserve"/> </xsd:restriction> </xsd:simpleType>
PIC S9 DISPLAY PIC S99 DISPLAY PIC S999 DISPLAY PIC S9999 DISPLAY	<xsd:simpleType> <xsd:restriction base="xsd:short"> <xsd:minInclusive value="-*n*"/> <xsd:maxInclusive value="*n*"/> </xsd:restriction> </xsd:simpleType> where *n* is the maximum value that can be represented by the pattern of '9' characters.
PIC S9(*z*) DISPLAY where $5 \leq z \leq 9$	<xsd:simpleType> <xsd:restriction base="xsd:int"> <xsd:minInclusive value="-*n*"/> <xsd:maxInclusive value="*n*"/> </xsd:restriction> </xsd:simpleType> where *n* is the maximum value that can be represented by the pattern of '9' characters.
PIC S9(*z*) DISPLAY where $9 < z$	<xsd:simpleType> <xsd:restriction base="xsd:long"> <xsd:minInclusive value="-*n*"/> <xsd:maxInclusive value="*n*"/> </xsd:restriction> </xsd:simpleType> where *n* is the maximum value that can be represented by the pattern of '9' characters.
PIC 9 DISPLAY PIC 99 DISPLAY PIC 999 DISPLAY PIC 9999 DISPLAY	<xsd:simpleType> <xsd:restriction base="xsd:unsignedShort"> <xsd:minInclusive value="0"/> <xsd:maxInclusive value="*n*"/> </xsd:restriction> </xsd:simpleType> where *n* is the maximum value that can be represented by the pattern of '9' characters.
PIC 9(*z*) DISPLAY where $5 \leq z \leq 9$	<xsd:simpleType> <xsd:restriction base="xsd:unsignedInt"> <xsd:minInclusive value="0"/> <xsd:maxInclusive value="*n*"/> </xsd:restriction> </xsd:simpleType> where *n* is the maximum value that can be represented by the pattern of '9' characters.

COBOL data description	Schema simpleType
PIC 9(*z*) DISPLAY where 9 < *z*	<xsd:simpleType> <xsd:restriction base="xsd:unsignedLong"> <xsd:minInclusive value="0"/> <xsd:maxInclusive value="*n*"/> </xsd:restriction> </xsd:simpleType> where *n* is the maximum value that can be represented by the pattern of '9' characters.
PIC S9(*n*) COMP PIC S9(*n*) COMP-4 PIC S9(*n*) COMP-5 PIC S9(*n*) BINARY where *n* ≤ 4.	<xsd:simpleType> <xsd:restriction base="xsd:short"> </xsd:restriction> </xsd:simpleType>
PIC S9(*n*) COMP PIC S9(*n*) COMP-4 PIC S9(*n*) COMP-5 PIC S9(*n*) BINARY where 5 ≤ *n* ≤ 9.	<xsd:simpleType> <xsd:restriction base="xsd:int"> </xsd:restriction> </xsd:simpleType>
PIC S9(*n*) COMP PIC S9(*n*) COMP-4 PIC S9(*n*) COMP-5 PIC S9(*n*) BINARY where 9 <*n*.	<xsd:simpleType> <xsd:restriction base="xsd:long"> </xsd:restriction> </xsd:simpleType>
PIC 9(*n*) COMP PIC 9(*n*) COMP-4 PIC 9(*n*) COMP-5 PIC 9(*n*) BINARY where *n* ≤ 4.	<xsd:simpleType> <xsd:restriction base="xsd:unsignedShort"> </xsd:restriction> </xsd:simpleType>
PIC 9(*n*) COMP PIC 9(*n*) COMP-4 PIC 9(*n*) COMP-5 PIC 9(*n*) BINARY where 5 ≤ *n* ≤ 9.	<xsd:simpleType> <xsd:restriction base="xsd:unsignedInt"> </xsd:restriction> </xsd:simpleType>
PIC 9(*n*) COMP PIC 9(*n*) COMP-4 PIC 9(*n*) COMP-5 PIC 9(*n*) BINARY where 9 <*n*.	<xsd:simpleType> <xsd:restriction base="xsd:unsignedLong"> </xsd:restriction> </xsd:simpleType>
PIC S9(*m*)V9(*n*) COMP-3	<xsd:simpleType> <xsd:restriction base="xsd:decimal"> <xsd:totalDigits value="*p*"/> <xsd:fractionDigits value="*n*"/> </xsd:restriction> </xsd:simpleType> where *p* = *m* + *n*.

COBOL data description	Schema simpleType
PIC 9(*m*)V9(*n*) COMP-3	<xsd:simpleType> <xsd:restriction base="xsd:decimal"> <xsd:totalDigits value="*p*"/> <xsd:fractionDigits value="*n*"/> <xsd:minInclusive value="0"/> </xsd:restriction> </xsd:simpleType> where *p* = *m* + *n*.
PIC S9(*m*) COMP-3 Supported at mapping level 3.0 when DATETIME=PACKED15	<xsd:simpleType> <xsd:restriction base="xsd:dateTime" </xsd:restriction> </xsd:simpleType> The format of the timestamp is CICS ABSTIME.
PIC S9(*m*)V9(*n*) DISPLAY Supported at mapping level 1.2 and higher	<xsd:simpleType> <xsd:restriction base="xsd:decimal"> <xsd:totalDigits value="*p*"/> <xsd:fractionDigits value="*n*"/> </xsd:restriction> </xsd:simpleType> where *p* = *m* + *n*.
COMP-1 Supported at mapping level 1.2 and higher	<xsd:simpleType> <xsd:restriction base="xsd:float"> </xsd:restriction> </xsd:simpletype>
COMP-2 Supported at mapping level 1.2 and higher	<xsd:simpleType> <xsd:restriction base="xsd:double"> </xsd:restriction> </xsd:simpletype>

Related reference:

"XML schema to COBOL mapping"
The DFHSC2LS and DFHWS2LS utility programs support mappings between XML schema definitions and COBOL data structures.

"How to handle variably repeating content in COBOL" on page 450
In COBOL, you cannot process variably repeating content by using pointer arithmetic to address each instance of the data. Other programming languages do not have this limitation. This example shows you how to handle variably repeating content in COBOL for a Web service application.

XML schema to COBOL mapping

The DFHSC2LS and DFHWS2LS utility programs support mappings between XML schema definitions and COBOL data structures.

The CICS assistants generate unique and valid names for COBOL variables from the schema element names using the following rules:

1. COBOL reserved words are prefixed with 'X'.

   For example, DISPLAY becomes XDISPLAY.

2. Characters other than A-Z, a-z, 0-9, or hyphen are replaced with 'X'.

   For example, monthly_total becomes monthlyXtotal. You can use the **MAPPING-OVERRIDES** parameter to change the way other characters are handled.

| For example, if you set the value UNDERSCORES-AS-HYPHENS, any underscore in
| the XML is converted to a hyphen instead of an X. So monthly_total becomes
| monthly-total.

3. If the last character is a hyphen, it is replaced with 'X'.

   For example, ca-request- becomes ca-requestX.

4. If the schema specifies that the variable has varying cardinality (that is, minOccurs and maxOccurs are specified on an xsd:element with different values), and the schema element name is longer than 23 characters, it is truncated to that length.

   If the schema specifies that the variable has fixed cardinality and the schema element name is longer than 28 characters, it is truncated to that length.

5. Duplicate names in the same scope are made unique by the addition of one or two numeric digits to the second and subsequent instances of the name.

   For example, three instances of year become year, year1, and year2.

6. Five characters are reserved for the strings -cont or -num, which are used when the schema specifies that the variable has varying cardinality; that is, when minOccurs and maxOccurs are specified with different values.

   For more information, see "Variable arrays of elements" on page 435.

7. For attributes, the previous rules are applied to the element name. The prefix attr- is added to the element name, and is followed by -value or -exist. If the total length is longer than 28 characters, the element name is truncated. For more information, see "Support for XML attributes" on page 439.

   The nillable attribute has special rules. The prefix attr- is added, but nil- is also added to the beginning of the element name. The element name is followed by -value. If the total length is longer than 28 characters, the element name is truncated.

The total length of the resulting name is 30 characters or less.

DFHSC2LS and DFHWS2LS map schema types to COBOL data description elements by using the specified mapping level according to the following table. Note the following points:

- If the **MAPPING-LEVEL** parameter is set to 1.2 or higher and the **CHAR-VARYING** parameter is set to NULL, variable-length character data is mapped to null-terminated strings and an extra character is allocated for the null-terminator.
- If the **MAPPING-LEVEL** parameter is set to 1.2 or higher and the **CHAR-VARYING** parameter is set to YES, variable-length character data is mapped to two related elements: a length field and a data field. For example:

```
<xsd:simpleType name="VariableStringType">
   <xsd:restriction base="xsd:string">
       <xsd:minLength value="1"/>
       <xsd:maxLength value="10000"/>
   </xsd:restriction>
</xsd:simpleType>
<xsd:element name="textString" type="tns:VariableStringType"/>
```

  maps to:

```
15 textString-length PIC S9999 COMP-5 SYNC
15 textString        PIC X(10000)
```

Schema simple type	COBOL data description
<xsd:simpleType> <xsd:restriction base="xsd:anyType"> </xsd:restriction> </xsd:simpleType>	Mapping level 2.0 and below: Not supported Mapping level 2.1: Supported
<xsd:simpleType> <xsd:restriction base="xsd:anySimpletype"> </xsd:restriction> </xsd:simpleType>	Mapping level 1.0: Not supported Mapping level 1.1 and higher: PIC X(255)
<xsd:simpleType> <xsd:restriction base="xsd:*type*" <xsd:length value="*z*"/> </xsd:restriction> </xsd:simpleType> where *type* is one of: string normalizedString token Name NMTOKEN language NCName ID IDREF ENTITY hexBinary	All mapping levels: PIC X(*z*)
<xsd:simpleType> <xsd:restriction base="xsd:*type*" </xsd:restriction> </xsd:simpleType> where *type* is one of: duration date time gDay gMonth gYear gMonthDay gYearMonth	All mapping levels: PIC X(32)
<xsd:simpleType> <xsd:restriction base="xsd:dateTime" </xsd:restriction> </xsd:simpleType>	Mapping level 1.2 and below: PIC X(32) Mapping level 2.0 and higher: PIC X(40) Mapping level 3.0 and higher: PIC S9(15) COMP-3 The format is CICS ABSTIME.

Schema simple type	COBOL data description

<xsd:simpleType>
  <xsd:restriction base="xsd:*type*">
  </xsd:restriction>
</xsd:simpleType>
 	
where *type* is one of:
  byte
  unsignedByte
	All mapping levels:
	PIC X DISPLAY

<xsd:simpleType> <xsd:restriction base="xsd:short"> </xsd:restriction> </xsd:simpleType>	All mapping levels: PIC S9999 COMP-5 SYNC or PIC S9999 DISPLAY
<xsd:simpleType> <xsd:restriction base="xsd:unsignedShort"> </xsd:restriction> </xsd:simpleType>	All mapping levels: PIC 9999 COMP-5 SYNC or PIC 9999 DISPLAY
<xsd:simpleType> <xsd:restriction base="xsd:integer"> </xsd:restriction> </xsd:simpleType>	All mapping levels: PIC S9(18) COMP-3
<xsd:simpleType> <xsd:restriction base="xsd:int"> </xsd:restriction> </xsd:simpleType>	All mapping levels: PIC S9(9) COMP-5 SYNC or PIC S9(9) DISPLAY
<xsd:simpleType> <xsd:restriction base="xsd:unsignedInt"> </xsd:restriction> </xsd:simpleType>	All mapping levels: PIC 9(9) COMP-5 SYNC or PIC 9(9) DISPLAY
<xsd:simpleType> <xsd:restriction base="xsd:long"> </xsd:restriction> </xsd:simpleType>	All mapping levels: PIC S9(18) COMP-5 SYNC or PIC S9(18) DISPLAY
<xsd:simpleType> <xsd:restriction base="xsd:unsignedLong"> </xsd:restriction> </xsd:simpleType>	All mapping levels: PIC 9(18) COMP-5 SYNC or PIC 9(18) DISPLAY
<xsd:simpleType> <xsd:restriction base="xsd:decimal"> <xsd:totalDigits value="*m*" <xsd:fractionDigits value="*n*" </xsd:restriction> </xsd:simpleType>	All mapping levels: PIC 9(*p*)V9(*n*) COMP-3 where $p = m - n$.
<xsd:simpleType> <xsd:restriction base="xsd:boolean"> </xsd:restriction> </xsd:simpleType>	All mapping levels: PIC X DISPLAY The value x'00' implies false, x'01' implies true.
<xsd:simpleType> <xsd:list> <xsd:simpleType> <xsd:restriction base="xsd:int"/> </xsd:simpleType> </xsd:list> </xsd:simpleType>	Mapping level 1.0: Not supported Mapping level 1.1 and higher: PIC X(255)

Schema simple type	COBOL data description
<xsd:simpleType> <xsd:union memberTypes="xsd:int xsd:string"/> </xsd:simpleType>	Mapping level 1.0: Not supported Mapping level 1.1 and higher: PIC X(255)
<xsd:simpleType> <xsd:restriction base="xsd:base64Binary"> <xsd:length value="z"/> </xsd:restriction> </xsd:simpleType> <xsd:simpleType> <xsd:restriction base="xsd:base64Binary"> </xsd:restriction> </xsd:simpleType> where the length is not defined.	Mapping level 1.0: Not supported Mapping level 1.1: PIC X(*y*) where *y* =4×(ceil(*z*/3)). ceil(*x*) is the smallest integer greater than or equal to *x*. Mapping level 1.2 and higher: PIC X(*z*) where the length is fixed. PIC X(16) where the length is not defined. The field holds the 16-byte name of the container that stores the binary data.
<xsd:simpleType> <xsd:restriction base="xsd:float"> </xsd:restriction> </xsd:simpletype>	Mapping level 1.1 and below: PIC X(32) Mapping level 1.2 and higher: COMP-1
<xsd:simpleType> <xsd:restriction base="xsd:double"> </xsd:restriction> </xsd:simpletype>	Mapping level 1.1 and below: PIC X(32) Mapping level 1.2 and higher: COMP-2

Some of the schema types shown in the table map to a COBOL format of COMP-5 SYNC or of DISPLAY, depending on the values (if any) that are specified in the `minInclusive` and `maxInclusive` facets:

- For signed types (`short`, `int`, and `long`), DISPLAY is used when the following are specified:

  ```
  <xsd:minInclusive value="-a"/>
  <xsd:maxInclusive value="a"/>
  ```

  where *a* is a string of '9's.
- For unsigned types (`unsignedShort`, `unsignedInt`, and `unsignedLong`), DISPLAY is used when the following are specified:

  ```
  <xsd:minInclusive value="0"/>
  <xsd:maxInclusive value="a"/>
  ```

  where *a* is a string of '9's.

When any other value is specified, or no value is specified, COMP-5 SYNC is used.

Related reference:

"COBOL to XML schema mapping" on page 411
The DFHLS2SC and DFHLS2WS utility programs support mappings between COBOL data structures and XML schema definitions.

"How to handle variably repeating content in COBOL" on page 450
In COBOL, you cannot process variably repeating content by using pointer arithmetic to address each instance of the data. Other programming languages do not have this limitation. This example shows you how to handle variably repeating content in COBOL for a Web service application.

"Support for <xsd:any> and xsd:anyType" on page 442
DFHWS2LS and DFHSC2LS support the use of <xsd:any> and `xsd:anyType` in the XML schema. You can use the <xsd:any> XML schema element to describe a section of an XML document with undefined content. `xsd:anyType` is the base data type from which all simple and complex data types are derived; it has no restrictions or constraints on the data content.

C and C++ to XML schema mapping

The DFHLS2SC and DFHLS2WS utility programs support mappings between C and C++ data types and XML schema definitions.

C and C++ names are converted to XML names according to the following rules:

1. Characters that are not valid in XML element names are replaced with `'X'`.

   For example, `monthly-total` becomes `monthlyXtotal`.
2. Duplicate names are made unique by the addition of one or more numeric digits.

   For example, two instances of `year` become `year` and `year1`.

DFHLS2SC and DFHLS2WS map C and C++ data types to schema elements according to the following table. C and C++ types that are not shown in the table are not supported by DFHLS2SC or DFHLS2WS. The `_Packed` qualifier is supported for structures. These restrictions apply:

- Header files must contain a top level `struct` instance.
- You cannot declare a structure type that contains itself as a member.
- The following C and C++ data types are not supported:
  ```
  decimal
  long double
  wchar_t (C++ only)
  ```
- The following characters are ignored if they are present in the header file.
  Storage class specifiers:
  ```
  auto
  register
  static
  extern
  mutable
  ```
  Qualifiers
  ```
  const
  volatile
  _Export (C++ only)
  ```
  Function specifiers
  ```
  inline (C++ only)
  virtual (C++ only)
  ```
  Initial values

- The header file must not contain these items:
  Unions
  Class declarations
  Enumeration data types
  Pointer type variables
  Template declarations
  Predefined macros; that is, macros with names that start and end with two underscore characters (__)
  The line continuation sequence (a \ symbol that is immediately followed by a newline character)
  Prototype function declarators
  Preprocessor directives
  Bit fields
  The `__cdecl` (or `_cdecl`) keyword (C++ only)
- The application programmer must use a 32-bit compiler to ensure that an `int` maps to 4 bytes.
- The following C++ reserved keywords are not supported:
  `explicit`
  `using`
  `namespace`
  `typename`
  `typeid`
- If the **MAPPING-LEVEL** parameter is set to 1.2 or higher and the **CHAR-VARYING** parameter is set to NULL, character arrays are mapped to an `xsd:string` and are processed as null-terminated strings.
- If the **MAPPING-LEVEL** parameter is set to 1.2 or higher and the **CHAR-VARYING** parameter is set to BINARY, character arrays are mapped to `xsd:base64Binary` and are processed as binary data.
- If the **MAPPING-LEVEL** parameter is set to 1.2 or higher and the **CHAR-VARYING** parameter is set to COLLAPSE, `<xsd:whiteSpace value="collapse"/>` is generated for strings.

C and C++ data type	Schema simpleType

`char[z]`	`<xsd:simpleType>`
	`  <xsd:restriction base="xsd:string">`
	`  <xsd:length value="z"/>`
	`  </xsd:restriction>`
	`</xsd:simpletype>`

`char[8]`
 	
Supported at mapping level 3.0 and higher when DATETIME=PACKED15
	`<xsd:simpleType>`
	`  <xsd:restriction base="xsd:dateTime"`
	`  </xsd:restriction>`
	`</xsd:simpleType>`
	
	The format of the time stamp is CICS ABSTIME.

`char`	`<xsd:simpleType>`
	`  <xsd:restriction base="xsd:byte">`
	`  </xsd:restriction>`
	`</xsd:simpletype>`

C and C++ data type	Schema simpleType

`unsigned char`	`<xsd:simpleType>`
	`  <xsd:restriction base="xsd:unsignedByte">`
	`  </xsd:restriction>`
	`</xsd:simpletype>`

`short`	`<xsd:simpleType>`
	`  <xsd:restriction base="xsd:short">`
	`  </xsd:restriction>`
	`</xsd:simpletype>`

`unsigned short`	`<xsd:simpleType>`
	`  <xsd:restriction base="xsd:unsignedShort">`
	`  </xsd:restriction>`
	`</xsd:simpletype>`

`int`
`long`
	`<xsd:simpleType>`
	`  <xsd:restriction base="xsd:int">`
	`  </xsd:restriction>`
	`</xsd:simpletype>`

`unsigned int`
`unsigned long`
	`<xsd:simpleType>`
	`  <xsd:restriction base="xsd:unsignedInt">`
	`  </xsd:restriction>`
	`</xsd:simpletype>`

`long long`	`<xsd:simpleType>`
	`  <xsd:restriction base="xsd:long">`
	`  </xsd:restriction>`
	`</xsd:simpletype>`

`unsigned long long`	`<xsd:simpleType>`
	`  <xsd:restriction base="xsd:unsignedLong">`
	`  </xsd:restriction>`
	`</xsd:simpletype>`

`bool`
 	
(C++ only)
	`<xsd:simpleType>`
	`  <xsd:restriction base="xsd:boolean">`
	`  </xsd:restriction>`
	`</xsd:simpletype>`

`float`
 	
Supported at mapping level 1.2 and higher
	`<xsd:simpleType>`
	`  <xsd:restriction base="xsd:float">`
	`  </xsd:restriction>`
	`</xsd:simpletype>`

`double`
 	
Supported at mapping level 1.2 and higher
	`<xsd:simpleType>`
	`  <xsd:restriction base="xsd:double">`
	`  </xsd:restriction>`
	`</xsd:simpletype>`

Related reference:

"XML schema to C and C++ mapping"
The DFHSC2LS and DFHWS2LS utility programs support mappings between the XML schema definitions that are included in each Web service description and C and C++ data types.

XML schema to C and C++ mapping

The DFHSC2LS and DFHWS2LS utility programs support mappings between the XML schema definitions that are included in each Web service description and C and C++ data types.

The CICS assistants generate unique and valid names for C and C++ variables from the schema element names using the following rules:

1. Characters other than A-Z, a-z, 0-9, or _ are replaced with 'X'.

For example, `monthly-total` becomes `monthlyXtotal`.

2. If the first character is not an alphabetic character, it is replaced by a leading 'X'.

   For example, `_monthlysummary` becomes `Xmonthlysummary`.

3. If the schema element name is longer than 50 characters, it is truncated to that length.
4. Duplicate names in the same scope are made unique by the addition of one or more numeric digits.

   For example, two instances of `year` become `year` and `year1`.

5. Five characters are reserved for the strings `_cont` or `_num`, which are used when the schema specifies that the variable has varying cardinality; that is, when `minOccurs` and `maxOccurs` are specified on an `xsd:element`.

   For more information, see "Variable arrays of elements" on page 435.

6. For attributes, the previous rules are applied to the element name. The prefix `attr_` is added to the element name, and it is followed by `_value` or `_exist`. If the total length is longer than 28 characters, the element name is truncated.

   The nillable attribute has special rules. The prefix `attr_` is added, but `nil_` is also added to the beginning of the element name. The element name is followed by `_value`. If the total length is longer than 28 characters, the element name is truncated.

The total length of the resulting name is 57 characters or less.

DFHSC2LS and DFHWS2LS map schema types to C and C++ data types according to the following table. The following rules also apply:

- If the **MAPPING-LEVEL** parameter is set to 1.2 or higher and the **CHAR-VARYING** parameter is set to NULL, variable-length character data is mapped to null-terminated strings and an extra character is allocated for the null-terminator.
- If the **MAPPING-LEVEL** parameter is set to 1.2 or higher and the **CHAR-VARYING** parameter is set to YES, variable-length character data is mapped to two related elements: a length field and a data field.

Schema simpleType	C and C++ data type

`<xsd:simpleType>`
`  <xsd:restriction base="xsd:anyType">`
`  </xsd:restriction>`
`</xsd:simpleType>`
	Mapping level 2.0 and below:
	Not supported
	
	Mapping level 2.1 and higher:
	
	Supported

`<xsd:simpleType>`
`  <xsd:restriction base="xsd:anySimpletype">`
`  </xsd:restriction>`
`</xsd:simpleType>`
	Mapping level 1.0:
	Not supported
	
	Mapping level 1.1 and higher:
	
	`char[255]`

Schema simpleType	C and C++ data type
<xsd:simpleType> <xsd:restriction base="xsd:*type*"> <xsd:length value="*z*"/> </xsd:restriction> </xsd:simpleType> where *type* is one of: string normalizedString token Name NMTOKEN language NCName ID IDREF ENTITY hexBinary	All mapping levels: char[*z*]
<xsd:simpleType> <xsd:restriction base="xsd:*type*"> </xsd:restriction> </xsd:simpleType> where *type* is one of: duration date decimal time gDay gMonth gYear gMonthDay gYearMonth	All mapping levels: char[32]
<xsd:simpleType> <xsd:restriction base="xsd:dateTime"> </xsd:restriction> </xsd:simpleType>	Mapping level 1.2 and below: char[32] Mapping level 2.0 and higher: char[40] Mapping level 3.0 and higher: char[8] The format of the time stamp is CICS ABSTIME.
<xsd:simpleType> <xsd:restriction base="xsd:byte"> </xsd:restriction> </xsd:simpletype>	All mapping levels: signed char
<xsd:simpleType> <xsd:restriction base="xsd:unsignedByte"> </xsd:restriction> </xsd:simpletype>	All mapping levels: char

Schema simpleType	C and C++ data type
<xsd:simpleType> <xsd:restriction base="xsd:short"> </xsd:restriction> </xsd:simpletype>	All mapping levels: short
<xsd:simpleType> <xsd:restriction base="xsd:unsignedShort"> </xsd:restriction> </xsd:simpletype>	All mapping levels: unsigned short
<xsd:simpleType> <xsd:restriction base="xsd:integer"> </xsd:restriction> </xsd:simpletype>	All mapping levels: char[33]
<xsd:simpleType> <xsd:restriction base="xsd:int"> </xsd:restriction> </xsd:simpletype>	All mapping levels: int
<xsd:simpleType> <xsd:restriction base="xsd:unsignedInt"> </xsd:restriction> </xsd:simpletype>	All mapping levels: unsigned int
<xsd:simpleType> <xsd:restriction base="xsd:long"> </xsd:restriction> </xsd:simpletype>	All mapping levels: long long
<xsd:simpleType> <xsd:restriction base="xsd:unsignedLong"> </xsd:restriction> </xsd:simpletype>	All mapping levels: unsigned long long
<xsd:simpleType> <xsd:restriction base="xsd:boolean"> </xsd:restriction> </xsd:simpletype>	All mapping levels: bool (C++ only) short (C only)
<xsd:simpleType> <xsd:list> <xsd:simpleType> <xsd:restriction base="xsd:int"/> </xsd:simpleType> </xsd:list> </xsd:simpleType>	Mapping level 1.0: Not supported Mapping level 1.1 and higher: char[255]
<xsd:simpleType> <xsd:union memberTypes="xsd:int xsd:string"/> </xsd:simpleType>	Mapping level 1.0: Not supported Mapping level 1.1 and higher: char[255]

Schema simpleType	C and C++ data type
<xsd:simpleType> <xsd:restriction base="xsd:base64Binary"> <xsd:length value="z"/> </xsd:restriction> </xsd:simpleType>	Mapping level 1.1 and below: char[*y*] where *y* =4×(ceil(*z*/3)). ceil(*x*) is the smallest integer greater than or equal to *x*.
<xsd:simpleType> <xsd:restriction base="xsd:base64binary"> </xsd:restriction> </xsd:simpletype> where the length is not defined	Mapping level 1.2 and higher: char[*z*] where the length is fixed. char[16] is the name of the container that stores the binary data when the length is not defined.
<xsd:simpleType> <xsd:restriction base="xsd:float"> </xsd:restriction> </xsd:simpletype>	Mapping level 1.1 and below: char[32] Mapping level 1.2 and higher: float(*)
<xsd:simpleType> <xsd:restriction base="xsd:double"> </xsd:restriction> </xsd:simpletype>	Mapping level 1.0 and below: char[32] Mapping level 1.2 and higher: double(*)

Related reference:

"C and C++ to XML schema mapping" on page 420
The DFHLS2SC and DFHLS2WS utility programs support mappings between C and C++ data types and XML schema definitions.

"Support for <xsd:any> and xsd:anyType" on page 442
DFHWS2LS and DFHSC2LS support the use of <xsd:any> and xsd:anyType in the XML schema. You can use the <xsd:any> XML schema element to describe a section of an XML document with undefined content. xsd:anyType is the base data type from which all simple and complex data types are derived; it has no restrictions or constraints on the data content.

PL/I to XML schema mapping

The DFHLS2SC and DFHLS2WS utility programs support mappings between PL/I data structures and XML schema definitions. Because the Enterprise PL/I compiler and older PL/I compilers differ, two language options are supported: PLI-ENTERPRISE and PLI-OTHER.

PL/I names are converted to XML names according to the following rules:

1. Characters that are not valid in XML element names are replaced with 'x'.

   For example, monthly$total becomes monthlyxtotal.
2. Duplicate names are made unique by the addition of one or more numeric digits.

   For example, two instances of year become year and year1.

DFHLS2SC and DFHLS2WS map PL/I data types to schema elements according to the following table. PL/I types that are not shown in the table are not supported by DFHLS2SC or DFHLS2WS. The following restrictions also apply:

- Data items with the COMPLEX attribute are not supported.
- Data items with the FLOAT attribute are supported at a mapping level of 1.2 or higher. Enterprise PL/I FLOAT IEEE is not supported.
- VARYING and VARYINGZ pure DBCS strings are supported at a mapping level of 1.2 or higher.
- Data items specified as DECIMAL(*p*,*q*) are supported only when $p \geq q$
- Data items specified as BINARY(*p*,*q*) are supported only when $q = 0$.
- If the PRECISION attribute is specified for a data item, it is ignored.
- PICTURE strings are not supported.
- ORDINAL data items are treated as FIXED BINARY(7) data types.
- If the **MAPPING-LEVEL** parameter is set to 1.2 or higher and the **CHAR-VARYING** parameter is set to NULL, character arrays are mapped to an `xsd:string` and are processed as null-terminated strings; this mapping does not apply for Enterprise PL/I.
- If the **MAPPING-LEVEL** parameter is set to 1.2 or higher and the **CHAR-VARYING** parameter is set to BINARY, character arrays are mapped to `xsd:base64Binary` and are processed as binary data.
- If the **MAPPING-LEVEL** parameter is set to 1.2 or higher and the **CHAR-VARYING** parameter is set to COLLAPSE, `<xsd:whiteSpace value="collapse"/>` is generated for strings.

DFHLS2SC and DFHLS2WS do not fully implement the padding algorithms of PL/I; therefore, you must declare padding bytes explicitly in your data structure. DFHLS2SC and DFHLS2WS issue a message if they detect that padding bytes are missing. Each top-level structure must start on a double-word boundary and each byte in the structure must be mapped to the correct boundary. Consider this code fragment:

```
3 FIELD1 FIXED BINARY(7),
3 FIELD2 FIXED BINARY(31),
3 FIELD3 FIXED BINARY(63);
```

In this example:

- FIELD1 is 1 byte long and can be aligned on any boundary.
- FIELD2 is 4 bytes long and must be aligned on a full word boundary.
- FIELD3 is 8 bytes long and must be aligned on a double word boundary.

The Enterprise PL/I compiler aligns the fields in the following order:

1. FIELD3 is aligned first because it has the strongest boundary requirements.
2. FIELD2 is aligned at the fullword boundary immediately before FIELD3.
3. FIELD1 is aligned at the byte boundary immediately before FIELD3.

Finally, so that the entire structure will be aligned at a fullword boundary, the compiler inserts three padding bytes immediately before FIELD1.

Because DFHLS2WS does not insert equivalent padding bytes, you must declare them explicitly before the structure is processed by DFHLS2WS. For example:

```
3 PAD1   FIXED BINARY(7),
3 PAD2   FIXED BINARY(7),
3 PAD3   FIXED BINARY(7),
3 FIELD1 FIXED BINARY(7),
3 FIELD2 FIXED BINARY(31),
3 FIELD3 FIXED BINARY(63);
```

Alternatively, you can change the structure to declare all the fields as unaligned and recompile the application that uses the structure. For further information on PL/I structural memory alignment requirements, refer to *Enterprise PL/I Language Reference*.

PL/I data description	Schema

`FIXED BINARY (n)`
where $n \leq 7$
	`<xsd:simpleType>`
	`  <xsd:restriction base="xsd:byte"/>`
	`</xsd:simpleType>`

`FIXED BINARY (n)`
 	
where $8 \leq n \leq 15$
	`<xsd:simpleType>`
	`  <xsd:restriction base="xsd:short"/>`
	`</xsd:simpleType>`

`FIXED BINARY (n)`
 	
where $16 \leq n \leq 31$
	`<xsd:simpleType>`
	`  <xsd:restriction base="xsd:int"/>`
	`</xsd:simpleType>`

`FIXED BINARY (n)`
 	
where $32 \leq n \leq 63$
Restriction: Enterprise PL/I only
	`<xsd:simpleType>`
	`  <xsd:restriction base="xsd:long"/>`
	`</xsd:simpleType>`

`UNSIGNED FIXED BINARY(n)`
 	
where $n \leq 8$
Restriction: Enterprise PL/I only
	`<xsd:simpleType>`
	`  <xsd:restriction base="xsd:unsignedByte"/>`
	`</xsd:simpleType>`

`UNSIGNED FIXED BINARY(n)`
 	
where $9 \leq n \leq 16$
Restriction: Enterprise PL/I only
	`<xsd:simpleType>`
	`  <xsd:restriction base="xsd:unsignedShort"/>`
	`</xsd:simpleType>`

`UNSIGNED FIXED BINARY(n)`
 	
where $17 \leq n \leq 32$
Restriction: Enterprise PL/I only
	`<xsd:simpleType>`
	`  <xsd:restriction base="xsd:unsignedInt"/>`
	`</xsd:simpleType>`

`UNSIGNED FIXED BINARY(n)`
 	
where $33 \leq n \leq 64$
Restriction: Enterprise PL/I only
	`<xsd:simpleType>`
	`  <xsd:restriction base="xsd:unsignedLong"/>`
	`</xsd:simpleType>`

`FIXED DECIMAL(n,m)`	`<xsd:simpleType>`
	`  <xsd:restriction base="xsd:decimal">`
	`  <xsd:totalDigits value="n"/>`
	`  <xsd:fractionDigits value="m"/>`
	`  </xsd:restriction>`
	`</xsd:simpleType>`

`FIXED DECIMAL(15)`
 	
Supported at mapping level 3.0 and higher when DATETIME=PACKED15
	`<xsd:simpleType>`
	`  <xsd:restriction base="xsd:dateTime"`
	`  </xsd:restriction>`
	`</xsd:simpleType>`
	
	The format of the time stamp is CICS ABSTIME.

PL/I data description	Schema
BIT(*n*) where *n* is a multiple of 8. Other values are not supported.	<xsd:simpleType> <xsd:restriction base="xsd:hexBinary"> <xsd:length value="*m*"/> </xsd:restriction> </xsd:simpleType> where *m* = *n*/8
CHARACTER(*n*) VARYING and VARYINGZ are also supported at mapping level 1.2 and higher. **Restriction:** VARYINGZ is supported only by Enterprise PL/I	<xsd:simpleType> <xsd:restriction base="xsd:string"> <xsd:maxLength value="*n*"/> <xsd:whiteSpace value="preserve"/> </xsd:restriction> </xsd:simpleType>
GRAPHIC(*n*) VARYING and VARYINGZ are also supported at mapping level 1.2 and higher. **Restriction:** VARYINGZ is supported only by Enterprise PL/I	<xsd:simpleType> <xsd:restriction base="xsd:hexBinary"> <xsd:length value="*m*"/> </xsd:restriction> </xsd:simpleType> at a mapping level of 1.0 and 1.1, where *m* = 2**n* <xsd:simpleType> <xsd:restriction base="xsd:string"> <xsd:length value="*n*"/> <xsd:whiteSpace value="preserve"/> </xsd:restriction> </xsd:simpleType> at a mapping level of 1.2 or higher
WIDECHAR(*n*) **Restriction:** Enterprise PL/I only	<xsd:simpleType> <xsd:restriction base="xsd:hexBinary"> <xsd:length value="*m*"/> </xsd:restriction> </xsd:simpleType> at a mapping level of 1.0 and 1.1, where *m* = 2**n* <xsd:simpleType> <xsd:restriction base="xsd:hexBinary"> <xsd:length value="*n*"/> </xsd:restriction> </xsd:simpleType> at a mapping level of 1.2 or higher
ORDINAL **Restriction:** Enterprise PL/I only	<xsd:simpleType> <xsd:restriction base="xsd:byte"/> </xsd:simpleType>
BINARY FLOAT(*n*) where *n* <= 21 Supported at mapping level 1.2 and higher.	<xsd:simpleType> <xsd:restriction base="xsd:float"> </xsd:restriction> </xsd:simpletype>
BINARY FLOAT(*n*) where 21 < *n* <= 53 Values greater than 53 are not supported. Supported at mapping level 1.2 and higher.	<xsd:simpleType> <xsd:restriction base="xsd:double"> </xsd:restriction> </xsd:simpletype>

PL/I data description	Schema
DECIMAL FLOAT(*n*)where *n* <= 6 Supported at mapping level 1.2 and higher.	<xsd:simpleType> <xsd:restriction base="xsd:float"> </xsd:restriction> </xsd:simpletype>
DECIMAL FLOAT(*n*)where 6 < *n* <= 16 Values greater than 16 are not supported. Supported at mapping level 1.2 and higher.	<xsd:simpleType> <xsd:restriction base="xsd:double"> </xsd:restriction> </xsd:simpletype>

Related reference:

"XML schema to PL/I mapping"
The DFHSC2LS and DFHWS2LS utility programs support mappings between XML schema definitions and PL/I data structures. Because the Enterprise PL/I compiler and older PL/I compilers differ, two language options are supported: PLI-ENTERPRISE and PLI-OTHER.

XML schema to PL/I mapping

The DFHSC2LS and DFHWS2LS utility programs support mappings between XML schema definitions and PL/I data structures. Because the Enterprise PL/I compiler and older PL/I compilers differ, two language options are supported: PLI-ENTERPRISE and PLI-OTHER.

The CICS assistants generate unique and valid names for PL/I variables from the schema element names using the following rules:

1. Characters other than A-Z, a-z, 0-9, @, #, or $ are replaced with 'X'.

   For example, monthly-total becomes monthlyXtotal.
2. If the schema specifies that the variable has varying cardinality (that is, minOccurs and maxOccurs attributes are specified with different values on the xsd:element), and the schema element name is longer than 24 characters, it is truncated to that length.

   If the schema specifies that the variable has fixed cardinality and the schema element name is longer than 29 characters, it is truncated to that length.
3. Duplicate names in the same scope are made unique by the addition of one or more numeric digits to the second and subsequent instances of the name.

   For example, three instances of year become year, year1, and year2.
4. Five characters are reserved for the strings _cont or _num, which are used when the schema specifies that the variable has varying cardinality; that is, when minOccurs and maxOccurs attributes are specified with different values.

   For more information, see "Variable arrays of elements" on page 435.
5. For attributes, the previous rules are applied to the element name. The prefix attr- is added to the element name and is followed by -value or -exist. If the total length is longer than 28 characters, the element name is truncated. For more information, see "Support for XML attributes" on page 439.

   The nillable attribute has special rules. The prefix attr- is added, but nil- is also added to the beginning of the element name. The element name is followed by -value. If the total length is longer than 28 characters, the element name is truncated.

The total length of the resulting name is 31 characters or less.

DFHSC2LS and DFHWS2LS map schema types to PL/I data types according to the following table. Also note the following points:

- If the **MAPPING-LEVEL** parameter is set to 1.2 or higher and the **CHAR-VARYING** parameter is set to NULL, variable-length character data is mapped to null-terminated strings and an extra character is allocated for the null-terminator.
- If the **MAPPING-LEVEL** parameter is set to 1.2 or higher and the **CHAR-VARYING** parameter is not specified, by default variable-length character data is mapped to a VARYINGZ data type for Enterprise PL/I and VARYING data type for Other PL/I.
- Variable-length binary data is mapped to a VARYING data type if it less than 32 768 bytes and to a container if it is more than 32 768 bytes.

Schema	PL/I data description
<xsd:simpleType> <xsd:restriction base="xsd:anyType"> </xsd:restriction> </xsd:simpleType>	Mapping level 2.0 and below: Not supported Mapping level 2.1 and higher: Supported
<xsd:simpleType> <xsd:restriction base="xsd:anySimpletype"> </xsd:restriction> </xsd:simpleType>	Mapping level 1.1 and higher:CHAR(255)
<xsd:simpleType> <xsd:restriction base="xsd:*type*"> <xsd:maxLength value="*z*"/> <xsd:whiteSpace value="preserve"/> </xsd:restriction> </xsd:simpleType> where *type* is one of: string normalizedString token Name NMTOKEN language NCName ID IDREF ENTITY	All mapping levels:CHARACTER(*z*)
<xsd:simpleType> <xsd:restriction base="xsd:*type*"> </xsd:restriction> </xsd:simpleType> where *type* is one of: duration date time gDay gMonth gYear gMonthDay gYearMonth	All mapping levels:CHAR(32)

Schema	PL/I data description
<xsd:simpleType> <xsd:restriction base="xsd:dateTime"> </xsd:restriction> </xsd:simpleType>	Mapping level 1.2 and below: CHAR(32) Mapping level 2.0 and higher: CHAR(40) Mapping level 3.0 and higher: FIXED DECIMAL(15) The format of the time stamp is CICS ABSTIME.
<xsd:simpleType> <xsd:restriction base="xsd:hexBinary"> <xsd:length value="*y*"/> </xsd:restriction> </xsd:simpleType>	Mapping level 1.1 and below: BIT(*z*) where $z = 8 \times y$ and $z < 4095$ bytes. CHAR(*z*) where $z = 8 \times y$ and $z > 4095$ bytes. Mapping levels 1.2 and higher: CHAR(*y*)
<xsd:simpleType> <xsd:restriction base="xsd:byte"> </xsd:restriction> </xsd:simpleType>	All mapping levels: **Enterprise PL/I** SIGNED FIXED BINARY (7) **Other PL/I** FIXED BINARY (7)
<xsd:simpleType> <xsd:restriction base="xsd:unsignedByte"> </xsd:restriction> </xsd:simpleType>	All mapping levels: **Enterprise PL/I** UNSIGNED FIXED BINARY (8) **Other PL/I** FIXED BINARY (8)
<xsd:simpleType> <xsd:restriction base="xsd:short"> </xsd:restriction> </xsd:simpleType>	All mapping levels: **Enterprise PL/I** SIGNED FIXED BINARY (15) **Other PL/I** FIXED BINARY (15)
<xsd:simpleType> <xsd:restriction base="xsd:unsignedShort"> </xsd:restriction> </xsd:simpleType>	All mapping levels: **Enterprise PL/I** UNSIGNED FIXED BINARY (16) **Other PL/I** FIXED BINARY (16)
<xsd:simpleType> <xsd:restriction base="xsd:integer"> </xsd:restriction> </xsd:simpleType>	All mapping levels: **Enterprise PL/I** FIXED DECIMAL(31,0) **Other PL/I** FIXED DECIMAL(15,0)

Schema	PL/I data description
<xsd:simpleType> <xsd:restriction base="xsd:int"> </xsd:restriction> </xsd:simpleType>	All mapping levels: **Enterprise PL/I** SIGNED FIXED BINARY (31) **Other PL/I** FIXED BINARY (31)
<xsd:simpleType> <xsd:restriction base="xsd:unsignedInt"> </xsd:restriction> </xsd:simpleType>	Mapping level 1.1 and below: **Enterprise PL/I** UNSIGNED FIXED BINARY(32) Mapping level 1.2 and higher: **Enterprise PL/I** CHAR(*y*) where *y* is a fixed length that is less than 16 MB. All mapping levels: **Other PL/I** BIT(64)
<xsd:simpleType> <xsd:restriction base="xsd:long"> </xsd:restriction> </xsd:simpleType>	Mapping level 1.1 and below: **Enterprise PL/I** SIGNED FIXED BINARY(63) Mapping level 1.2 and higher: **Enterprise PL/I** CHAR(*y*) where *y* is a fixed length that is less than 16 MB. All mapping levels: **Other PL/I** BIT(64)
<xsd:simpleType> <xsd:restriction base="xsd:unsignedLong"> </xsd:restriction> </xsd:simpleType>	Mapping level 1.1 and below: **Enterprise PL/I** UNSIGNED FIXED BINARY(64) Mapping level 1.2 and higher: **Enterprise PL/I** CHAR(*y*) where *y* is a fixed length that is less than 16 MB. All mapping levels: **Other PL/I** BIT(64)

Schema	PL/I data description
<xsd:simpleType> <xsd:restriction base="xsd:boolean"> </xsd:restriction> </xsd:simpleType>	Mapping level 1.1 and below: **Enterprise PL/I** SIGNED FIXED BINARY (7) **Other PL/I** FIXED BINARY (7) Mapping level 1.2 and higher: **Enterprise PL/I** BIT(7) BIT(1) **Other PL/I** BIT(7) BIT(1) where BIT(7) is provided for alignment and BIT(1) contains the Boolean mapped value.
<xsd:simpleType> <xsd:restriction base="xsd:decimal"> <xsd:totalDigits value="*n*"/> <xsd:fractionDigits value="*m*"/> </xsd:restriction> </xsd:simpleType>	All mapping levels:FIXED DECIMAL(*n*,*m*)
<xsd:simpleType> <xsd:list> <xsd:simpleType> <xsd:restriction base="xsd:int"/> </xsd:simpleType> </xsd:list> </xsd:simpleType>	All mapping levels:CHAR(255)
<xsd:simpleType> <xsd:union memberTypes="xsd:int xsd:string"/> </xsd:simpleType>	All mapping levels:CHAR(255)
<xsd:simpleType> <xsd:restriction base="xsd:base64Binary"> <xsd:length value="*y*"/> </xsd:restriction> </xsd:simpleType> <xsd:simpleType> <xsd:restriction base="xsd:base64Binary"> </xsd:restriction> </xsd:simpleType> where the length is not defined	Mapping level 1.0: Not supported Mapping level 1.1: CHAR(*z*) where *z* =4×(ceil(*y*/3)). ceil(*x*) is the smallest integer greater than or equal to *x*. Mapping level 1.2 and higher: CHAR(*y*) where the length is fixed. CHAR(16) where the length is not defined. The field holds the 16-byte name of the container that stores the binary data.

Schema	PL/I data description
<xsd:simpleType> <xsd:restriction base="xsd:float"> </xsd:restriction> </xsd:simpletype>	Mapping levels 1.0 and 1.1: CHAR(32) Mapping level 1.2 and higher: **Enterprise PL/I** DECIMAL FLOAT(6) HEXADEC **Other PL/I** DECIMAL FLOAT(6)
<xsd:simpleType> <xsd:restriction base="xsd:double"> </xsd:restriction> </xsd:simpletype>	Mapping levels 1.0 and 1.1: CHAR(32) Mapping level 1.2 and higher: **Enterprise PL/I** DECIMAL FLOAT(16) HEXADEC **Other PL/I** DECIMAL FLOAT(16)

Related reference:

"PL/I to XML schema mapping" on page 426
The DFHLS2SC and DFHLS2WS utility programs support mappings between PL/I data structures and XML schema definitions. Because the Enterprise PL/I compiler and older PL/I compilers differ, two language options are supported: PLI-ENTERPRISE and PLI-OTHER.

"Support for <xsd:any> and xsd:anyType" on page 442
DFHWS2LS and DFHSC2LS support the use of <xsd:any> and xsd:anyType in the XML schema. You can use the <xsd:any> XML schema element to describe a section of an XML document with undefined content. xsd:anyType is the base data type from which all simple and complex data types are derived; it has no restrictions or constraints on the data content.

Variable arrays of elements

XML can contain an array with varying numbers of elements. In general, WSDL documents and XML schemas that contain varying numbers of elements do not map efficiently into a single high-level language data structure. CICS uses container-based mappings or inline mappings to handle varying numbers of elements in XML.

An array with a varying number of elements is represented in the XML schema by using the minOccurs and maxOccurs attributes on the element declaration:

- The minOccurs attribute specifies the minimum number of times that the element can occur. It can have a value of 0 or any positive integer.
- The maxOccurs attribute specifies the maximum number of times that the element can occur. It can have a value of any positive integer greater than or equal to the value of the minOccurs attribute. It can also take a value of unbounded, which indicates that no upper limit applies to the number of times the element can occur.
- The default value for both attributes is 1.

This example denotes an 8-byte string that is optional; that is, it can occur never or once in the application XML or SOAP message:

```
<xsd:element name="component" minOccurs="0" maxOccurs="1">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:length value="8"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

The following example denotes an 8-byte string that must occur at least once:

```
<xsd:element name="component" minOccurs="1" maxOccurs="unbounded">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:length value="8"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

In general, WSDL documents that contain varying numbers of elements do not map efficiently into a single high-level language data structure. Therefore, to handle these cases, CICS uses a series of connected data structures that are passed to the application program in a series of containers. These structures are used as input and output from the application:

- When CICS transforms XML to application data, it populates these structures with the application data and the application reads them.
- When CICS transforms the application data to XML, it reads the application data in the structures that have been populated by the application.

The format of these data structures is best explained with a series of examples. The XML can be from a SOAP message or from an application. These examples use an array of simple 8-byte fields. However, the model supports arrays of complex data types and arrays of data types that contain other arrays.

Fixed number of elements

The first example illustrates an element that occurs exactly three times:

```
<xsd:element name="component" minOccurs="3" maxOccurs="3">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:length value="8"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

In this example, because the number of times that the element occurs is known in advance, it can be represented as a fixed-length array in a simple COBOL declaration (or the equivalent in other languages):

```
05 component PIC X(8) OCCURS 3 TIMES
```

Varying number of elements at mapping level 2 and below

This example illustrates a mandatory element that can occur from one to five times:

```
<xsd:element name="component" minOccurs="1" maxOccurs="5">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:length value="8"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

The main data structure contains a declaration of two fields. When CICS transforms the XML to binary data, the first field `component-num` contains the number of times that the element appears in the XML, and the second field, `component-cont`, contains the name of a container:

```
05 component-num PIC S9(9) COMP-5
05 component-cont PIC X(16)
```

A second data structure contains the declaration of the element itself:

```
01 DFHWS-component
  02 component PIC X(8)
```

You must examine the value of `component-num` (which will contain a value in the range 1 to 5) to find out how many times the element occurs. The element contents are in the container named in `component-cont`; the container holds an array of elements, where each element is mapped by the `DFHWS-component` data structure.

If `minOccurs="0"` and `maxOccurs="1"`, the element is optional. To process the data structure in your application program, you must examine the value of `component-num`:

- If it is zero, the message has no component element and the contents of `component-cont` is undefined.
- If it is one, the component element is in the container named in `component-cont`.

The contents of the container are mapped by the `DFHWS-component` data structure.

Note: If the SOAP message consists of a single recurring element, DFHWS2LS generates two language structures. The main language structure contains the number of elements in the array and the name of a container which holds the array of elements. The second language structure maps a single instance of the recurring element.

Varying number of elements at mapping level 2.1 and above

At mapping level 2.1 and above, you can use the **INLINE-MAXOCCURS-LIMIT** parameter in the CICS assistants. The **INLINE-MAXOCCURS-LIMIT** parameter specifies the way that varying numbers of elements are handled. The mapping options for varying numbers of elements are container-based mapping, described in "Varying number of elements at mapping level 2 and below" on page 436, or inline mapping. The *value* of this parameter can be a positive integer in the range 0 - 32767:

- The default value of **INLINE-MAXOCCURS-LIMIT** is 1, which ensures that optional elements are mapped inline.
- A value of `0` for the **INLINE-MAXOCCURS-LIMIT** parameter prevents inline mapping.
- If `maxOccurs` is less than or equal to the value of **INLINE-MAXOCCURS-LIMIT**, inline mapping is used.
- If `maxOccurs` is greater than the value of **INLINE-MAXOCCURS-LIMIT**, container-based mapping is used.

Mapping varying numbers of elements inline results in the generation of both an array, as happens with the fixed occurrence example above, and a counter. The `component-num` field indicates how many instances of the element are present, and these are pointed to by the array. For the example shown in "Varying number of elements at mapping level 2 and below" on page 436, when **INLINE-MAXOCCURS-LIMIT** is less than or equal to 5, the generated data structure is like this:

```
05 component-num PIC S9(9) COMP-5 SYNC.
05 component OCCURS 5 PIC X(8).
```

The first field, component-num, is identical to the output for the container-based mapping example in the previous section. The second field contains an array of length 5 which is large enough to contain the maximum number of elements that can be generated.

Inline mapping differs from container-based mapping, which stores the number of occurrences of the element and the name of the container where the data is placed, because it stores all the data in the current container. Storing the data in the current container will generally improve performance and make inline mapping preferable.

Nested variable arrays

Complex WSDL documents and XML schemas can contain variably recurring elements, which in turn contain variably recurring elements. In this case, the structure described extends beyond the two levels described in the examples.

This example illustrates an optional element called <component2> that is nested in a mandatory element called <component1>, where the mandatory element can occur from one to five times:

```
<xsd:element name="component1" minOccurs="1" maxOccurs="5">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="component2" minOccurs="0" maxOccurs="1">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:length value="8"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

The top-level data structure is exactly the same as in the previous examples:

```
05 component1-num PIC S9(9) COMP-5
05 component1-cont PIC X(16)
```

However, the second data structure contains these elements:

```
01 DFHWS-component1
  02 component2-num PIC S9(9) COMP-5
  02 component2-cont PIC X(16)
```

A third-level structure contains these elements:

```
01 DFHWS-component2
  02 component2 PIC X(8)
```

The number of occurrences of the outermost element <component1> is in component1-num.

The container named in component1-cont contains an array with that number of instances of the second data structure DFHWS-component1.

Each instance of component2-cont names a different container, each of which contains the data structure mapped by the third-level structure DFHWS-component2.

To illustrate this structure, consider the fragment of XML that matches the example:

```
<component1><component2>string1</component2></component1>
<component1><component2>string2</component2></component1>
<component1></component1>
```

<component1> occurs three times. The first two each contain an instance of <component2>; the third instance does not.

In the top-level data structure, component1-num contains a value of 3. The container named in component1-cont has three instances of DFHWS-component1:

1. In the first, component2-num has a value of 1, and the container named in component2-cont holds *string1*.
2. In the second, component2-num has a value of 1, and the container named in component2-cont holds *string2*.
3. In the third, component2-num has a value of 0, and the contents of component2-cont are undefined.

In this instance, the complete data structure is represented by four containers in all:

- The root data structure in container DFHWS-DATA
- The container named in component1-cont
- Two containers named in the first two instances of component2-cont

Optional structures and xsd:choice

DFHWS2LS and DFHSC2LS support the use of maxOccurs and minOccurs on <xsd:sequence>, <xsd:choice>, and <xsd:all> elements only at mapping level 2.1 and above, where the minOccurs and maxOccurs attributes are set to minOccurs="0" and maxOccurs="1".

The assistants generate mappings that treat these elements as though each child element in them is optional. When you implement an application with these elements, ensure that invalid combinations of options are not generated by the application. Each of the elements has its own count field in the generated languages structure, these fields must either all be set to "0" or all be set to"1". Any other combination of values is invalid, except for with <xsd:choice> elements.

<xsd:choice> elements indicate that only one of the options in the element can be used. It is supported at all mapping levels. The assistants handle each of the options in an <xsd:choice> as though it is in an <xsd:sequence> element with minOccurs="0" and maxOccurs="1". Take care when you implement an application using the <xsd:choice> element to ensure that invalid combinations of options are not generated by the application. Each of the elements has its own count field in the generated languages structure, exactly one of which must be set to '1' and the others must all be set to '0'. Any other combination of values is invalid, except when the <xsd:choice> element is itself optional, in which case it is valid for all the fields to be set to '0'.

Support for XML attributes

XML schemas can specify attributes that are allowed or required in XML. The CICS assistant utilities DFHWS2LS and DFHSC2LS ignore XML attributes by default. To process XML attributes that are defined in the XML schema, the value of the **MAPPING-LEVEL** parameter must be 1.1 or higher.

Optional attributes

Attributes can be optional or required and can be associated with any element in a SOAP message or XML for an application. For every optional attribute defined in the schema, two fields are generated in the appropriate language structure:

1. An existence flag; this field is treated as a Boolean data type and is typically 1 byte in length.
2. A value; this field is mapped in the same way as an equivalently typed XML element. For example, an attribute of type `NMTOKEN` is mapped in the same way as an XML element of type `NMTOKEN`.

The attribute existence and value fields appear in the generated language structure before the field for the element with which they are associated. Unexpected attributes that appear in the instance document are ignored.

For example, consider the following schema attribute definition:

```
<xsd:attribute name="age" type="xsd:short" use="optional" />
```

This optional attribute maps to the following COBOL structure:

```
05 attr-age-exist  PIC X DISPLAY
05 attr-age-value  PIC S9999 COMP-5 SYNC
```

Runtime processing of optional attributes

The following runtime processing takes place for optional attributes:

- If the attribute is present, the existence flag is set and the value is mapped.
- If the attribute is not present, the existence flag is not set.
- If the attribute has a default value and is present, the value is mapped.
- If the attribute has a default value and is not present, the default value is mapped.

Optional attributes that have default values are treated as required attributes.

When CICS transforms the data to XML, the following runtime processing takes place:

- If the existence flag is set, the attribute is transformed and included in the XML.
- If the existence flag is not set, the attribute is not included in the XML.

Required attributes and runtime processing

For every attribute that is required, only the value field is generated in the appropriate language structure.

If the attribute is present in the XML, the value is mapped. If the attribute is not present, the following processing occurs:

- If the application is a Web service provider, CICS generates a SOAP fault message indicating an error in the client SOAP message.
- If the application is a Web service requester, CICS issues a message and returns a conversion error response with a RESP2 code of 13 to the application.
- If the application is using the **TRANSFORM XMLTODATA** command, CICS issues a message and returns an invalid request response with a RESP2 code of 3 to the application.

When CICS produces a SOAP message based on the contents of a COMMAREA or container, the attribute is transformed and included in the message. When an application uses the **TRANSFORM DATATOXML** command, CICS also transforms the attribute and includes it in the XML.

The nillable attribute

The nillable attribute is a special attribute that can appear on an `xsd:element` in an XML schema. It specifies that the `xsi:nil` attribute is valid for the element in XML. If an element has the `xsi:nil` attribute specified, it indicates that the element is present but has no value, and therefore no content is associated with it.

If an XML schema has defined the nillable attribute as true, it is mapped as a required attribute that takes a Boolean value.

When CICS receives a SOAP message or has to transform XML for an application that contains an `xsi:nil` attribute, the value of the attribute is true or false. If the value is true, the application must ignore the values of the element or nested elements in the scope of the `xsi:nil` attribute.

When CICS produces a SOAP message or XML based on the contents of a COMMAREA or container for which the value for the `xsi:nil` attribute is true, the following processing occurs:

- The `xsi:nil` attribute is generated into the XML or SOAP message.
- The value of the associated element is ignored.
- Any nested elements within the element are ignored.

SOAP message example

Consider the following example XML schema, which could be part of a WSDL document:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="root" nillable="true">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element nillable="true" name="num" type="xsd:int"  maxOccurs="3" minOccurs="3"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Here is an example of a partial SOAP message that conforms to this schema:

```
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <num xsi:nil="true"/>
 <num>15</num>
 <num xsi:nil="true"/>
</root>
```

In COBOL, this SOAP message maps to these elements:

```
05    root
10    attr-nil-root-value  PIC X DISPLAY
10    num                  OCCURS 3
15    num1                 PIC S9(9) COMP-5 SYNC
15    attr-nil-num-value   PIC X DISPLAY
10    filler               PIC X(3)
```

Related reference:

"Mapping levels for the CICS assistants" on page 404
A mapping is the set of rules that specifies how information is converted between language structures and XML schemas. To benefit from the most sophisticated mappings available, you are recommended to set the **MAPPING-LEVEL** parameter in the CICS assistants to the latest level.

Support for <xsd:any> and xsd:anyType

DFHWS2LS and DFHSC2LS support the use of <xsd:any> and xsd:anyType in the XML schema. You can use the <xsd:any> XML schema element to describe a section of an XML document with undefined content. xsd:anyType is the base data type from which all simple and complex data types are derived; it has no restrictions or constraints on the data content.

Before you can use <xsd:any> and xsd:anyType with the CICS assistants, set the following parameters:

- Set the **MAPPING-LEVEL** parameter to 2.1 or higher.
- For a Web service provider application, set the **PGMINT** parameter to CHANNEL.

<xsd:any> example

This example uses an <xsd:any> element to describe some optional unstructured XML content following the "Surname" tag in the "Customer" tag:

```
<xsd:element name="Customer">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element name="Title" type="xsd:string"/>
   <xsd:element name="FirstName" type="xsd:string"/>
   <xsd:element name="Surname" type="xsd:string"/>
   <xsd:any minOccurs="0"/>
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>
```

An example SOAP message that conforms to this XML schema is:

```
<xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <Customer xmlns="http://www.example.org/anyExample">
      <Title xmlns="">Mr</Title>
      <FirstName xmlns="">John</FirstName>
      <Surname xmlns="">Smith</Surname>
      <ExtraInformation xmlns="http://www.example.org/ExtraInformation">
        <!-- This 'ExtraInformation' tag is associated with the optional xsd:any from the XML schema.
             It can contain any well formed XML. -->
        <ExampleField1>one</ExampleField1>
        <ExampleField2>two</ExampleField2>
      </ExtraInformation>
    </Customer>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

If this SOAP message is sent to CICS, CICS populates the Customer-xml-cont container with the following XML data:

```
<ExtraInformation xmlns="http://www.example.org/ExtraInformation">
  <!-- This 'ExtraInformation' tag is associated with the optional xsd:any from the XML schema.
       It can contain any well formed XML. -->
  <ExampleField1>one</ExampleField1>
  <ExampleField2>two</ExampleField2>
</ExtraInformation>
```

CICS also populates the `Customer-xmlns-cont` container with the following XML namespace declarations that are in scope; these declarations are separated by a space:

```
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns="http://www.example.org/anyExample"
```

`xsd:anyType` example

The `xsd:anyType` is the base data type from which all simple and complex data types are derived. It does not restrict the data content. If you do not specify a data type, it defaults to `xsd:anyType`; for example, these two XML fragments are equivalent:

```
<xsd:element name="Name" type="xsd:anyType"/>
```

```
<xsd:element name="Name"/>
```

Generated language structures

The language structures generated for <xsd:any> or `xsd:anyType` take the following form in COBOL and an equivalent form for the other languages:

elementName-xml-cont PIC X(16)
The name of a container that holds the raw XML. When CICS processes an incoming SOAP message, it places the subset of the SOAP message that the <xsd:any> or `xsd:anyType` defines into this container. The application can process the XML data only natively. The application must generate the XML, populate this container, and supply the container name.

This container must be populated in text mode. If CICS populates this container, it does so using the same variant of EBCDIC as the Web service is defined to use. Characters that do not exist in the target EBCDIC code page are replaced with substitute characters, even if the container is read by the application in UTF-8.

elementName-xmlns-cont PIC X(16)
The name of a container that holds any namespace prefix declarations that are in scope. The contents of this container are similar to those of the DFHWS-XMLNS container, except that it includes all the namespace declarations that are in scope and that are relevant, rather than only the subset from the SOAP Envelope tag.

This container must be populated in text mode. If CICS populates this container, it does so using the same variant of EBCDIC as the Web service is defined to use. Characters that do not exist in the target EBCDIC code page are replaced with substitute characters, even if the container is read by the application in UTF-8.

This container is used only when processing SOAP messages sent to CICS. If the application tries to supply a container with namespace declarations when an output SOAP message is generated, the container and its contents are ignored by CICS. CICS requires that the XML supplied by the application is entirely self-contained with respect to namespace declarations.

The name of the XML element that contains the <xsd:any> element is included in the variable names that are generated for the <xsd:any> element. In the <xsd:any> example, the <xsd:any> element is nested inside the <xsd:element name="Customer"> element and the variable names that are generated for the <xsd:any> element are `Customer-xml-cont PIC X(16)` and `Customer-xmlns-cont PIC X(16)`.

For an `xsd:anyType` type, the direct XML element name is used; in the `xsd:anyType` example above, the variable names are `Name-xml-cont PIC X(16)` and `Name-xmlns-cont PIC X(16)`.

Related tasks:

"Handling <xsd:any> data types" on page 464
If you are working with an XML schema that contains one or more <xsd:any> data types, the XML assistants can map the data type to a pair of CICS containers. You can write an application program to parse the XML in the containers.

Related reference:

"XML schema to COBOL mapping" on page 415
The DFHSC2LS and DFHWS2LS utility programs support mappings between XML schema definitions and COBOL data structures.

"XML schema to C and C++ mapping" on page 422
The DFHSC2LS and DFHWS2LS utility programs support mappings between the XML schema definitions that are included in each Web service description and C and C++ data types.

"XML schema to PL/I mapping" on page 430
The DFHSC2LS and DFHWS2LS utility programs support mappings between XML schema definitions and PL/I data structures. Because the Enterprise PL/I compiler and older PL/I compilers differ, two language options are supported: `PLI-ENTERPRISE` and `PLI-OTHER`.

Support for <xsd:choice>

An `<xsd:choice>` element indicates that only one of the options in the element can be used. The CICS assistants provide varying degrees of support for `<xsd:choice>` elements at the various mapping levels.

Support for <xsd:choice> at mapping level 2.2 and higher

At mapping level 2.2 and higher, DFHWS2LS and DFHSC2LS provide improved support for `<xsd:choice>` elements. The assistants generate a new container that stores the value associated with the `<xsd:choice>` element. The assistants generate language structures containing the name of a new container and an extra field:

fieldname-**enum**
: The discriminating field to indicate which of the options the `<xsd:choice>` element will use.

fieldname-**cont**
: The name of the container that stores the option to be used. A further language structure is generated to map the value of the option.

The following XML schema fragment includes an `<xsd:choice>` element:

```
<xsd:element name="choiceExample">
   <xsd:complexType>
      <xsd:choice>
         <xsd:element name="option1" type="xsd:string" />
         <xsd:element name="option2" type="xsd:int" />
```

```
        <xsd:element name="option3" type="xsd:short" maxOccurs="2" minOccurs="2" />
      </xsd:choice>
   </xsd:complexType>
</xsd:element>
```

If this XML schema fragment is processed at mapping level 2.2 or higher, the assistant generates the following COBOL language structures:

```
   03 choiceExample.
      06 choiceExample-enum         PIC X DISPLAY.
          88 empty                     VALUE X'00'.
          88 option1                   VALUE X'01'.
          88 option2                   VALUE X'02'.
          88 option3                   VALUE X'03'.
      06 choiceExample-cont         PIC X(16).


01 Example-option1.
   03 option1-length                PIC S9999 COMP-5 SYNC.
   03 option1                       PIC X(255).

01 Example-option2.
   03 option2                       PIC S9(9) COMP-5 SYNC.

01 Example-option3.
   03 option3 OCCURS 2             PIC S9999 COMP-5 SYNC.
```

Limitations for <xsd:choice> at mapping level 2.2 and higher

DFHSC2LS and DFHWS2LS do not support nested `<xsd:choice>` elements; for example, the following XML is not supported:

```
<xsd:choice>
   <xsd:element name ="name1" type="string"/>
   <xsd:choice>
      <xsd:element name ="name2a" type="string"/>
      <xsd:element name ="name2b" type="string"/>
   </xsd:choice>
</xsd:choice>
```

DFHSC2LS and DFHWS2LS do not support recurring `<xsd:choice>` elements; for example, the following XML is not supported:

```
<xsd:choice maxOccurs="2">
   <xsd:element name ="name1" type="string"/>
</xsd:choice>
```

DFHSC2LS and DFHWS2LS support a maximum of 255 options in an `<xsd:choice>` element.

Support for <xsd:choice> at mapping level 2.1 and below

At mapping level 2.1 and below, DFHWS2LS provides limited support for `<xsd:choice>` elements. DFHWS2LS treats each of the options in an `<xsd:choice>` element as though it is an `<xsd:sequence>` element that can occur at most once.

Only one of the options in an `<xsd:choice>` element can be used, so take care when you implement an application using the `<xsd:choice>` element that you generate only valid combinations of options. Each of the elements has its own `count` field in the generated languages structure, exactly one of which must be set to 1 and the others must all be set to 0. Any other combination of values is incorrect, except when the `<xsd:choice>` is itself optional, in which case it is valid for all of the fields to be set to 0.

Related reference:

"Support for <xsd:any> and xsd:anyType" on page 442
DFHWS2LS and DFHSC2LS support the use of <xsd:any> and xsd:anyType in the XML schema. You can use the <xsd:any> XML schema element to describe a section of an XML document with undefined content. xsd:anyType is the base data type from which all simple and complex data types are derived; it has no restrictions or constraints on the data content.

"Support for abstract elements and abstract data types" on page 448
The CICS assistants provide support for abstract elements and abstract data types at mapping level 2.2 and higher. The CICS assistants map abstract elements and abstract data types in a similar way to substitution groups.

"Support for substitution groups"
You can use a substitution group to define a group of XML elements that are interchangeable. The CICS assistants provide support for substitution groups at mapping level 2.2 and higher.

Support for <xsd:sequence>

The use of <xsd:sequence> elements with the CICS assistants is supported with a number of limitations.

The use of minOccurs and maxOccurs attributes are not supported for the <xsd:sequence> element. The exceptions to this rule are when minOccurs="0" and maxOccurs="1" or minOccurs="1" and maxOccurs="1".

The CICS assistants do not allow two elements with the same name to be used in the same <xsd:sequence> element. For example, the sequence {a, b, c, a} is rejected by the CICS Assistants. To avoid this limitation, replace the sequence with {a maxOccurs="2", b, c}.

The CICS Assistants allow only one <xsd:any> element, or one element that is treated as an <xsd:any>, in the same <xsd:sequence> element. Elements that are treated as <xsd:any> elements include abstract elements that do not have a substitution group defined.

When parsing XML, CICS treats <xsd:sequence> elements like <xsd:all> elements; this means that CICS does not check whether the order of the items in the sequence is correct. For example, if the schema defines a sequence of {a, b, c}, CICS tolerates XML that contains a, b, and c in any order. However; when CICS generates XML, the order of items in an <xsd:sequence> is always preserved.

CICS does not automatically detect missing XML data. For example, if an element in the <xsd:sequence> is defined as mandatory, (minOccurs="1" maxOccurs="1"), but does not occur in the XML document, CICS will report this problem only if runtime validation is enabled

Support for substitution groups

You can use a substitution group to define a group of XML elements that are interchangeable. The CICS assistants provide support for substitution groups at mapping level 2.2 and higher.

At mapping level 2.2 and higher, DFHSC2LS and DFHWS2LS support substitution groups using similar mappings to those used for <xsd:choice> elements. The assistant generates an enumeration field and a new container name in the language structure.

The following XML schema fragment includes an array of two `subGroupParent` elements, each of which can be replaced with `replacementOption1` or `replacementOption2`:

```
<xsd:element name="subGroupExample" >
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref="subGroupParent" maxOccurs="2" minOccurs="2" />
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>

<xsd:element name="subGroupParent" type="xsd:anySimpleType" />
<xsd:element name="replacementOption1" type="xsd:int" substitutionGroup="subGroupParent" />
<xsd:element name="replacementOption2" type="xsd:short" substitutionGroup="subGroupParent" />
```

Processing this XML fragment with the assistant generates the following COBOL language structures:

```
   03 subGroupExample.
      06 subGroupParent OCCURS2.
         09 subGroupExample-enum PIC X DISPLAY.
            88 empty              VALUE X '00'.
            88 replacementOption1  VALUE X '01'.
            88 replacementOption2  VALUE X '02'.
            88 subGroupParent      VALUE X '03'.
         09 subGroupExample-cont PIC X (16).



01 Example-replacementOption1.
   03 replacementOption1      PIC S9(9) COMP-5 SYNC.

01 Example-replacementOption2.
   03 replacementOption2      PIC S9999 COMP-5 SYNC.

01 Example-subGroupParent.
   03 subGroupParent-length   PIC S9999 COMP-5 SYNC.
   03 subGroupParent          PIC X(255).
```

For more information about substitution groups, see the *W3C XML Schema Part 1: Structures Second Edition specification*: http://www.w3.org/TR/xmlschema-1/#Elements_Equivalence_Class

Related reference:

"Support for <xsd:any> and xsd:anyType" on page 442
DFHWS2LS and DFHSC2LS support the use of <xsd:any> and xsd:anyType in the XML schema. You can use the <xsd:any> XML schema element to describe a section of an XML document with undefined content. xsd:anyType is the base data type from which all simple and complex data types are derived; it has no restrictions or constraints on the data content.

"Support for <xsd:choice>" on page 444
An <xsd:choice> element indicates that only one of the options in the element can be used. The CICS assistants provide varying degrees of support for <xsd:choice> elements at the various mapping levels.

"Support for abstract elements and abstract data types"
The CICS assistants provide support for abstract elements and abstract data types at mapping level 2.2 and higher. The CICS assistants map abstract elements and abstract data types in a similar way to substitution groups.

Support for abstract elements and abstract data types

The CICS assistants provide support for abstract elements and abstract data types at mapping level 2.2 and higher. The CICS assistants map abstract elements and abstract data types in a similar way to substitution groups.

Support for abstract elements at mapping level 2.2 and higher

At mapping level 2.2 and above, DFHSC2LS and DFHWS2LS treat abstract elements in almost the same way as substitution groups except that the abstract element is not a valid member of the group. If there are no substitutable elements, the abstract element is treated as an <xsd:any> element and uses the same mappings as an <xsd:any> element at mapping level 2.1.

The following XML schema fragment specifies two options that can be used in place of the abstract element. The abstract element itself is not a valid option:

```
<xsd:element name="abstractElementExample" >
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref="abstractElementParent" maxOccurs="2" minOccurs="2" />
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>

<xsd:element name="abstractElementParent" type="xsd:anySimpleType" abstract="true" />
<xsd:element name="replacementOption1" type="xsd:int" substitutionGroup="abstractElementParent" />
<xsd:element name="replacementOption2" type="xsd:short" substitutionGroup="abstractElementParent" />
```

Processing this XML fragment with the assistant generates the following COBOL language structures:

```
   03 abstractElementExample.
      06 abstractElementParent OCCURS 2.
         09 abstractElementExample-enum  PIC X DISPLAY.
            88 empty                          VALUE X '00'.
            88 replacementOption1             VALUE X '01'.
            88 replacementOption2             VALUE X '02'.
         09 abstractElementExample-cont  PIC X (16).


01 Example-replacementOption1.
   03 replacementOption1                  PIC S9(9) COMP-5 SYNC.
```

```
01 Example-replacementOption2.
   03 replacementOption2                PIC S9999 COMP-5 SYNC.
```

For more information about abstract elements, see the *W3C XML Schema Part 0: Primer Second Edition specification*: http://www.w3.org/TR/xmlschema-0/#SubsGroups

Support for abstract data types at mapping level 2.2 and higher

At mapping level 2.2 and higher, DFHSC2LS and DFHWS2LS treat abstract data types as substitution groups. The assistant generates an enumeration field and a new container name in the language structure.

The following XML schema fragment specifies two alternatives that can be used in place of the abstract type:

```
<xsd:element name="AbstractDataTypeExample" type="abstractDataType" />

<xsd:complexType name="abstractDataType" abstract="true">
   <xsd:simpleContent>
      <xsd:extension base="xsd:string" />
   </xsd:simpleContent>
</xsd:complexType>
<xsd:complexType name="option1">
   <xsd:simpleContent>
      <xsd:restriction base="abstractDataType">
         <xsd:length value="5" />
      </xsd:restriction>
   </xsd:simpleContent>
</xsd:complexType>
<xsd:complexType name="option2">
   <xsd:simpleContent>
      <xsd:restriction base="abstractDataType">
         <xsd:length value="10" />
      </xsd:restriction>
   </xsd:simpleContent>
</xsd:complexType>
```

Processing this XML fragment with the assistant generates the following COBOL language structures:

```
   03 AbstractDataTypeExamp-enum   PIC X DISPLAY.
     88 empty                         VALUE X'00'.
     88 option1                       VALUE X'01'.
     88 option2                       VALUE X'02'.
   03 AbstractDataTypeExamp-cont   PIC X(16).
```

The language structures are generated into separate copy books. The language structure generated for `option1` is generated into one copybook:

```
   03 option1          PIC X(5).
```

The language structure for `option2` is generated into a different copybook:

```
   03 option2          PIC X(10).
```

For more information about abstract data types, see the *W3C XML Schema Part 0: Primer Second Edition specification*: http://www.w3.org/TR/xmlschema-0/#SubsGroups

Related reference:

"Support for <xsd:any> and xsd:anyType" on page 442
DFHWS2LS and DFHSC2LS support the use of <xsd:any> and xsd:anyType in the XML schema. You can use the <xsd:any> XML schema element to describe a section of an XML document with undefined content. xsd:anyType is the base data type from which all simple and complex data types are derived; it has no restrictions or constraints on the data content.

"Support for <xsd:choice>" on page 444
An <xsd:choice> element indicates that only one of the options in the element can be used. The CICS assistants provide varying degrees of support for <xsd:choice> elements at the various mapping levels.

"Support for substitution groups" on page 446
You can use a substitution group to define a group of XML elements that are interchangeable. The CICS assistants provide support for substitution groups at mapping level 2.2 and higher.

How to handle variably repeating content in COBOL

In COBOL, you cannot process variably repeating content by using pointer arithmetic to address each instance of the data. Other programming languages do not have this limitation. This example shows you how to handle variably repeating content in COBOL for a Web service application.

This technique also applies to transforming XML to application data using the **TRANSFORM** API commands. The following example WSDL document represents a Web service with application data that consists of an 8-character string that recurs a variable number of times:

```
<?xml version="1.0"?>
<definitions name="ExampleWSDL"
             targetNamespace="http://www.example.org/variablyRepeatingData/"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:tns="http://www.example.org/variablyRepeatingData/"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <types>
   <xsd:schema targetNamespace="http://www.example.org/variablyRepeatingData/">
     <xsd:element name="applicationData">
       <xsd:complexType>
         <xsd:sequence>
           <xsd:element name="component" minOccurs="1" maxOccurs="unbounded">
             <xsd:simpleType>
               <xsd:restriction base="xsd:string">
                 <xsd:length value="8"/>
               </xsd:restriction>
             </xsd:simpleType>
           </xsd:element>
         </xsd:sequence>
       </xsd:complexType>
     </xsd:element>
   </xsd:schema>
 </types>

 <message name="exampleMessage">
   <part element="tns:applicationData" name="messagePart"/>
 </message>

 <portType name="examplePortType">
   <operation name="exampleOperation">
     <input message="tns:exampleMessage"/>
     <output message="tns:exampleMessage"/>
   </operation>
 </portType>
```

```
 <binding name="exampleBinding" type="tns:examplePortType">
   <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
   <operation name="exampleOperation">
     <soap:operation soapAction=""/>
     <input><soap:body parts="messagePart" encodingStyle="" use="literal"/></input>
     <output><soap:body parts="messagePart" encodingStyle="" use="literal"/></output>
   </operation>
 </binding>
</definitions>
```

Processing this WSDL document through DFHWS2LS generates the following COBOL language structures:

```
          03 applicationData.

            06 component-num                PIC S9(9) COMP-5 SYNC.
            06 component-cont               PIC X(16).


         01 DFHWS-component.
           03 component                    PIC X(8).
```

Note that the 8-character `component` field is defined in a separate structure called `DFHWS-component`. The main data structure is called `applicationData` and it contains two fields, `component-num` and `component-cont`. The `component-num` field indicates how many instances of the `component` data are present and the `component-cont` field indicates the name of a container that holds the concatenated list of `component` fields.

The following COBOL code demonstrates one way to process the list of variably recurring data. It makes use of a linkage section array to address subsequent instances of the data, each of which is displayed by using the `DISPLAY` statement:

```
IDENTIFICATION DIVISION.
       PROGRAM-ID.    EXVARY.

       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.

      * working storage variables
       01 APP-DATA-PTR             USAGE IS POINTER.
       01 APP-DATA-LENGTH          PIC S9(8) COMP.
       01 COMPONENT-PTR            USAGE IS POINTER.
       01 COMPONENT-DATA-LENGTH    PIC S9(8) COMP.
       01 COMPONENT-COUNT          PIC S9(8) COMP-4 VALUE 0.
       01 COMPONENT-LENGTH         PIC S9(8) COMP.

       LINKAGE SECTION.

      * a large linkage section array
       01 BIG-ARRAY PIC X(659999).

      * application data structures produced by DFHWS2LS
      * this is normally referenced with a COPY statement
       01 DFHWS2LS-data.
          03 applicationData.
            06 component-num  PIC S9(9) COMP-5 SYNC.
            06 component-cont PIC X(16).

       01 DFHWS-component.
          03 component        PIC X(8).
```

```
       PROCEDURE DIVISION USING DFHEIBLK.
       A-CONTROL SECTION.
       A010-CONTROL.

      * Get the DFHWS-DATA container
           EXEC CICS GET CONTAINER('DFHWS-DATA')
                     SET(APP-DATA-PTR)
                     FLENGTH(APP-DATA-LENGTH)
           END-EXEC
           SET ADDRESS OF DFHWS2LS-data TO APP-DATA-PTR

      * Get the recurring component data
           EXEC CICS GET CONTAINER(component-cont)
                     SET(COMPONENT-PTR)
                     FLENGTH(COMPONENT-DATA-LENGTH)
           END-EXEC

      * Point the component structure at the first instance of the data
           SET ADDRESS OF DFHWS-component TO COMPONENT-PTR

      * Store the length of a single component
           MOVE LENGTH OF DFHWS-component TO COMPONENT-LENGTH

      * process each instance of component data in turn
           PERFORM WITH TEST AFTER
                 UNTIL COMPONENT-COUNT = component-num

      * display the current instance of the data
              DISPLAY 'component value is: ' component

      * address the next instance of the component data
              SET ADDRESS OF BIG-ARRAY TO ADDRESS OF DFHWS-component
              SET ADDRESS OF DFHWS-component
                  TO ADDRESS OF BIG-ARRAY (COMPONENT-LENGTH + 1:1)
              ADD 1 TO COMPONENT-COUNT

      * end the loop
           END-PERFORM.

      * Point the component structure back at the first instance of
      * of the data, for any further processing we may want to perform
           SET ADDRESS OF DFHWS-component TO COMPONENT-PTR

      * return to CICS.

           EXEC CICS
                RETURN
           END-EXEC

           GOBACK.
```

The code above provides a generic solution to handling variably repeating content. The array, BIG-ARRAY, moves to the start of each component in turn and does not remain fixed at the start of the data. The component data structure is then moved to point at the first byte of the next component. COMPONENT-PTR can be used to recover the start position of the component data if required.

Here is an example SOAP message that conforms to the WSDL document:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
 <SOAP-ENV:Body>
   <applicationData xmlns="http://www.example.org/variablyRepeatingData/">
     <component xmlns="">VALUE1</component>
     <component xmlns="">VALUE2</component>
```

```
     <component xmlns="">VALUE3</component>
   </applicationData>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Here is the output produced by the COBOL program when it processes the SOAP message:

```
CPIH 20080115103151 component value is: VALUE1
CPIH 20080115103151 component value is: VALUE2
CPIH 20080115103151 component value is: VALUE3
```

Related reference:

"COBOL to XML schema mapping" on page 411
The DFHLS2SC and DFHLS2WS utility programs support mappings between COBOL data structures and XML schema definitions.

"XML schema to COBOL mapping" on page 415
The DFHSC2LS and DFHWS2LS utility programs support mappings between XML schema definitions and COBOL data structures.

Support for variable-length values and white space

You can customize the way in which variable-length values and white space are handled by using settings on the CICS assistants and by adding facets directly into the XML schema.

Typically, the CICS XML assistant and the CICS Web services assistant map data strings to fixed-length character arrays; these arrays require padding with spaces or nulls. Mapping variable-length values to fixed-length data arrays can be inefficient and waste storage. If the length of your data is variable, you are recommended to customize the way these mappings are handled.

If you are converting from a language structure to an XML schema or WSDL document, you are recommended to specify the `whiteSpace` and `maxLength` facets in your XML schema and to set the **CHAR-VARYING-LIMIT** parameter on the assistants.

If you are converting from an XML schema or WSDL document to a language structure, you are recommended to set an appropriate value for the **CHAR-VARYING** parameter on the assistants.

Note: Null characters ('x00') are not valid in XML documents. Any null characters from application data parsed by CICS are seen to signify the end of a string and the value is truncated. When CICS generates application data it does so according to the value of the **CHAR-VARYING** parameter. For example, if the **CHAR-VARYING=NULL** option is specified, variable-length strings generated by CICS are ended with a null character.

Mapping variable-length values from XML to language structures

Use facets in the XML schema or specify certain parameters on the CICS assistants to customize the way in which mappings between your XML schema or WSDL document and the language structure are handled.

XML data types can be restricted using facets. Use the length facets (`length`, `maxLength`, and `minLength`) and the `whiteSpace` facet to customize how variable-length data in your XML is handled.

`length` Used to specify that the data is of fixed length.

maxLength
Used to specify the maximum length for the data type. If this value is not set for a string-based data type, the maximum length is unbounded.

minLength
Used to specify the minimum length for the data type. If this value is not set for a string-based data type, the minimum length is 0.

whiteSpace
Used to specify how white space around a data value is handled. White space includes spaces, tabs, and new lines. The `whiteSpace` facet can be set to `preserve`, `replace`, or `collapse`:

- A value of `preserve` maintains any white space in the data value.
- A value of `replace` means that any tabs or new lines are replaced with the appropriate number of spaces.
- A value of `collapse` means that leading, trailing, and embedded white space is removed and that all tabs, new lines, and consecutive spaces are replaced with single space characters.

If the `whiteSpace` facet is not set, white space is preserved.

For more information about XML schema facets, see the W3C recommendation schema *XML Schema Part 2: Datatypes Second Edition* http://www.w3.org/TR/xmlschema-2/#facets

The following parameters on the CICS assistants, DFHSC2LS and DFHWS2LS, can be used to alter the way that variable-length data is mapped from the XML schema to the language structure. These parameters are available at mapping level 1.2 or higher.

DEFAULT-CHAR-MAXLENGTH
Specifies the default array length of character data in characters for mappings where no length is implied in the XML schema or WSDL document. The value of this parameter can be a positive integer in the range of 1 - 2 147 483 647.

However, you are recommended to specify the maximum character length that you want DFHSC2LS or DFHWS2LS to use directly in your XML schema or WSDL document with the `maxLength` facet. Specifying the maximum length directly in the XML schema or WSDL document avoids problems associated with having one global default applied to all string-based data types.

CHAR-VARYING-LIMIT
Specifies the maximum size of variable-length character data that is mapped to the language structure. If the character data is larger than the value specified in this parameter, it is mapped to a container and the container name is used in the generated language structure. The value can range from 0 to the default 32 767 bytes.

CHAR-VARYING
Specifies how variable-length character data is mapped. If you do not specify this parameter, the default mapping depends on the language specified. You can select these options:

- **CHAR-VARYING=NO** specifies that variable-length character data is mapped as fixed-length strings.
- **CHAR-VARYING=NULL** specifies that variable-length character data is mapped to null-terminated strings.

- **CHAR-VARYING=YES** specifies that variable-length character data is mapped to a CHAR VARYING data type in PL/I. In the COBOL, C, and C++ languages, variable-length character data is mapped to an equivalent representation that comprises two related elements: data-length and the data.

Setting **CHAR-VARYING=YES** typically results in the best performance.

Mapping variable-length values from language structures to XML

You can customize the way in which mappings between your language structure and the XML schema, or WSDL document are handled. Set the **CHAR-VARYING** parameter on DFHLS2SC or DFHLS2WS,to `COLLAPSE` or `NULL` to change the way that character arrays are generated.

Setting the **CHAR-VARYING=NULL** option tells CICS to add a null character at the end of each character array when generating XML.

Setting the **CHAR-VARYING=COLLAPSE** option tells CICS to automatically remove any trailing spaces from the end of character arrays when generating XML. This option is available only at mapping level 2.1 or higher and **CHAR-VARYING=COLLAPSE** is the default value at mapping level 2.1 or higher for all languages other than C and C++. When the XML is parsed, all leading, trailing, and embedded white space is removed.

For more information, see Support for white space and variable length values in CICS Web services (Technote).

Generating mappings from language structures

To create XML from application data or vice versa, you create the mappings to describe how CICS will transform the data and XML at run time. You can start from any application data record; for example, you can start with a COMMAREA, VSAM file, temporary storage queue, or a DB2 record.

Before you begin

Before you create the mappings, you must make sure that these preconditions have been completed:

- You must have a language structure that describes the application record in a partitioned data set. The language structure can be written in any of the high-level languages supported by the CICS XML assistant: COBOL, PL/I, C, and C++. If this mapping is for use with an Atom feed, and you are using any of the fields in your application record to provide metadata for the Atom entries (such as the name of an author), make sure that those fields are not nested in your language structure. You can have structures of nested fields within a field that provides the content for an Atom entry.
- You must configure the user ID under which DFHLS2SC runs to use z/OS UNIX.
- The user ID must have read permission to access the language structure and write permission to put the output in the appropriate directories on z/OS UNIX.
- You must allocate sufficient storage to the user ID for the ID to run Java. You can use any supported version of Java. By default, DFHLS2SC uses the Java version specified in the **JAVADIR** parameter.

About this task

Use the CICS XML assistant to create the data mappings for the application record. For each high-level language supported by the CICS XML assistant, a few types of data are restricted or not supported. The CICS XML assistant issues error messages about any unsupported items that it identifies in your language structure. The reference information for the CICS XML assistant lists the restrictions that apply to each high-level language.

Procedure

1. Run the DFHLS2SC batch job. DFHLS2SC has optional parameters that you select to meet your requirements, such as selecting a particular code page or a name space. Use the following parameters as a minimum:
   a. Specify the high-level language of your language structure in the **LANG** parameter.
   b. If you are deploying the data mappings in a bundle, specify the name of a bundle resource in the **BUNDLE** parameter. If you are creating an XML binding for an Atom feed, do not specify this parameter.
   c. Specify the mapping level in the **MAPPING-LEVEL** parameter. If you are creating an XML binding for an Atom feed, you must use a mapping level of 3.0. For other situations, although you can use any mapping level, to get the most advanced mapping options you are recommended to use a mapping level of 3.0.
   d. Optional: If you are creating an XML binding for an Atom feed, and your application data record contains timestamps in CICS ABSTIME format, specify the optional parameter **DATETIME=PACKED15** to map these fields as timestamps.
   e. Specify the location and the code page of the language structures that describe the application record in the **PDSMEM** and **PDSCP** parameters.
   f. Specify the name and location of the schema file in the **SCHEMA** parameter. The file extension is `.xsd`. If you are creating a bundle, do not specify a location. DFHLS2SC creates the XML schema, but not the directory structure, if the file does not already exist.
   g. Specify the name and location of the XML binding in the **XSDBIND** parameter. The file extension is `.xsdbind`. If you are creating a bundle, do not specify a location. DFHLS2SC creates the XML binding, but not the directory structure, if the file does not already exist.

      Tip: Put the XML binding and the schema in the same directory structure to enable validation. Validation can be useful when you are testing out your application in a development or test environment. If you are creating a bundle, CICS puts the files in the same directory for you.

   If you specify the **BUNDLE** parameter, the batch job creates a bundle directory structure on z/OS UNIX. The bundle directory has a META-INF subdirectory that contains the bundle manifest. The batch job also creates an XML schema and an XML binding in the bundle directory, using the file names that you specified for the **SCHEMA** and **XSDBIND** parameters. If you do not specify the **BUNDLE** parameter, the batch job creates the XML schema and XML binding only in the specified location.
2. Install the BUNDLE resource, or an ATOMSERVICE resource that specifies this XML binding. The BUNDLE and ATOMSERVICE resources dynamically create an XMLTRANSFORM resource, which defines the location of the XML schema and binding file.

Results

When you generate mappings from language structures, only one XML transformation is possible.

Example

The following example shows DFHLS2SC with the minimum set of parameters specified.

```
//LS2SC JOB  'accounting information',name,MSGCLASS=A
//  SET QT=''''
//JAVAPROG EXEC DFHLS2SC,
// TMPFILE=&QT.&SYSUID.&QT
//INPUT.SYSUT1 DD *
LANG=COBOL
BUNDLE=/u/exampleapp/bundle/test1
LOGFILE=/u/exampleapp/xsdbind/example.log
MAPPING-LEVEL=3.0
PDSLIB=//CICSHLQ.SDFHSAMP
PDSMEM=CPYBK2
XSDBIND=example.xsdbind
SCHEMA=example.xsd
/*
```

What to do next

Write an application program to transform the application data to XML and vice versa. You can use the same mappings for both transformations. If you have created an XML binding for an Atom feed, continue with the steps to set up your Atom feed.

Related concepts:

"The CICS XML assistant" on page 389
The CICS XML assistant is a set of batch utilities that can help you to transform XML into high-level language structures and vice versa. The assistant supports rapid deployment of applications that perform XML processing with the minimum amount of programming effort.

Related tasks:

"Transforming application data to XML" on page 460
You can write an application program to transform application data to XML.

"Transforming XML to application data" on page 461
You can write an application program to transform XML to application data. You can also query the XML before transforming it.

Related reference:

"DFHLS2SC: high-level language to XML schema conversion" on page 390
The DFHLS2SC cataloged procedure generates an XML schema and an XML binding file from a high-level language structure. Use DFHLS2SC when you want to create a CICS program that can parse or create XML. The job control statements for DFHLS2SC, its symbolic parameters, its input parameters and their descriptions are listed.

"High-level language and XML schema mapping" on page 408
Use the CICS assistants to generate mappings between high-level language structures and XML schemas or WSDL documents. The CICS assistants also generate XML schemas or WSDL documents from high-level language data structures, or vice-versa.

Generating mappings from an XML schema

To create application data from XML or vice versa that conforms to an existing XML schema, you create the mappings to describe how CICS will transform the data at run time. You can start from an XML schema or a WSDL document.

Before you begin

You must have a valid XML schema or WSDL document. Before you create the mappings, you must make sure that these preconditions have been completed:

- You must have a valid XML schema or WSDL document.
- You must configure the user ID under which DFHSC2LS runs to use UNIX System Services.
- The user ID must have read permission to access the XML schema or WSDL document and write permission to put the output in the appropriate directories on z/OS UNIX.
- The user ID must have a sufficiently large storage allocation to run Java. You can use any supported version of Java. By default, DFHWS2LS uses the Java version specified in the **JAVADIR** parameter.

About this task

Use the CICS XML assistant to create the data mappings for the XML schema.

Procedure

1. Run the DFHSC2LS batch job. DFHSC2LS has optional parameters that you can select to meet your requirements, such as selecting a particular code page or specifying how to handle variable-length character data. Use the following parameters as a minimum:
   a. Specify the location of your input file in the **WSDL** or **SCHEMA** parameter. You can either use a WSDL document or an XML schema. If your input file contains references to other schemas or documents on the Internet and the system uses a proxy server, specify the domain name or IP address and the port number of the proxy server.
   b. Specify the high-level language that you want to generate in the **LANG** parameter. The XML assistant supports COBOL, C, C++, and PL\I languages.
   c. If you are deploying the data mappings in a bundle, specify the name and location of a bundle in the **BUNDLE** parameter.

   The XML assistant creates a library of supported transformations in the XML binding. For each global element in the input file, the assistant creates a separate transformation.

   If you specify the **BUNDLE** parameter, the batch job creates a bundle directory structure on z/OS UNIX. The bundle directory has a META-INF subdirectory that contains the bundle manifest. The batch job also creates an XML binding in the bundle directory and places the language structures in the specified location. The XML assistant also places a copy of the input file in the bundle directory. If you do not specify the **BUNDLE** parameter, the batch job creates the language structures and XML binding only in the specified location.
2. Install the BUNDLE resource. The BUNDLE resource dynamically creates an XMLTRANSFORM resource, which defines the location of the XML schema or WSDL document, the XML binding, and the language structures.

Results

When you generate mappings from an XML schema, CICS generates a language structure for every global element that is present in the schema.

Example

The following example shows DFHSC2LS with the minimum set of parameters specified.

```
//SC2LS JOB  'accounting information',name,MSGCLASS=A
//  SET QT=''''
//JAVAPROG EXEC DFHSC2LS,
// TMPFILE=&QT.&SYSUID.&QT
//INPUT.SYSUT1 DD *
LANG=COBOL
BUNDLE=/u/exampleapp/bundle/test1
LOGFILE=/u/exampleapp/xsdbind/example.log
MAPPING-LEVEL=3.0
PDSLIB=//CICSHLQ.SDFHSAMP
PDSMEM=CPYBK2
XSDBIND=example.xsdbind
SCHEMA=example.xsd
/*
```

What to do next

Write an application program to transform the application data to XML or vice versa. You can use the same mappings for both transformations.

Related concepts:

"The CICS XML assistant" on page 389
The CICS XML assistant is a set of batch utilities that can help you to transform XML into high-level language structures and vice versa. The assistant supports rapid deployment of applications that perform XML processing with the minimum amount of programming effort.

Related tasks:

"Transforming application data to XML"
You can write an application program to transform application data to XML.

"Transforming XML to application data" on page 461
You can write an application program to transform XML to application data. You can also query the XML before transforming it.

Transforming application data to XML

You can write an application program to transform application data to XML.

Before you begin

You must have an enabled XMLTRANSFORM resource that defines the XML binding and XML schema.

About this task

The XML assistant generates the mappings in the XML binding. If you started with a language structure and used DFHLS2SC, only one transformation to the XML is possible. If you started with an XML schema, there might be many XML to language structure transformations so your application must select which XML element to generate.

Procedure

1. The application program must create a channel and place the data associated with the language structure into a bit-mode container on that channel.
2. Use the **TRANSFORM DATATOXML** API command to transform the data to XML:

   ```
   EXEC CICS TRANSFORM DATATOXML
        XMLTRANSFORM('MyXmlTransformName')
        CHANNEL('MyChannelName')
        DATCONTAINER('SourceContainerName')
        XMLCONTAINER('TargetContainerName')
   ```

   If the XMLTRANSFORM resource supports only a single transformation from the language structure, you do not have to specify the conversion type on the command. If many transformations are possible, add the following options to your application program:

   ```
   ELEMNAME(elementName) ELEMNAMELEN(elementNameLength)
   ```

   These additional options indicate the XML element into which the application data is transformed and placed in the output container.
3. Install the application program.

Results

When the application runs the **TRANSFORM DATATOXML** command, CICS checks the XMLTRANSFORM resource to find the mappings in the XML binding and transforms the application binary data to XML using the containers on the channel. The XML is placed in the container that the application specified on the XMLCONTAINER option. The XML conforms to the XML schema that is defined in the XMLTRANSFORM resource.

What to do next

You can also use the same mappings to transform XML to application data. For details, see "Transforming XML to application data."

Related tasks:

"Generating mappings from language structures" on page 455
To create XML from application data or vice versa, you create the mappings to describe how CICS will transform the data and XML at run time. You can start from any application data record; for example, you can start with a COMMAREA, VSAM file, temporary storage queue, or a DB2 record.

"Generating mappings from an XML schema" on page 458
To create application data from XML or vice versa that conforms to an existing XML schema, you create the mappings to describe how CICS will transform the data at run time. You can start from an XML schema or a WSDL document.

Transforming XML to application data

You can write an application program to transform XML to application data. You can also query the XML before transforming it.

Before you begin

You must have an enabled XMLTRANSFORM resource that defines the XML binding and XML schema.

About this task

The CICS XML assistant generates the mappings in the XML binding. If you started with a language structure and used DFHLS2SC, only one transformation from the XML is possible. If you started with an XML schema, there might be many XML to language structure transformations so your application must select which language structure to use as output.

Procedure

1. Create a channel and place the XML into a text mode container on that channel. If the application puts the XML in a BIT mode container, CICS tries to determine the encoding of the text data, but it can take longer to process the container and the encoding might not be correct.
2. Use the **TRANSFORM XMLTODATA** API command. If only one transform is possible for the XML, you can use the following command:

```
EXEC CICS TRANSFORM XMLTODATA
      XMLTRANSFORM('MyXmlTransformName')
      CHANNEL('MyChannelName')
      XMLCONTAINER('SourceContainerName')
      DATCONTAINER('TargetContainerName')
```

If more than one transformation is possible, use the following additional options:

```
EXEC CICS TRANSFORM XMLTODATA
      XMLTRANSFORM('MyXmlTransformName')
      CHANNEL('MyChannelName')
      XMLCONTAINER('SourceContainerName')
      DATCONTAINER('TargetContainerName')
      ELEMNAME(elementName) ELEMNAMELEN(elementNameLength)
```

In the second example, CICS returns the name of the element that was found in the ELEMNAME option. The application can then use the name of the element to determine which of the library of language structures to use to interpret the contents of the target container.

3. Install the application program.

Results

When the application runs the **TRANSFORM XMLTODATA** command, CICS uses the details in the XMLTRANSFORM resource to transform the XML to application binary data using the containers on the channel.

Example

What to do next

You can also use the same mappings to transform application data to XML. For details, see "Transforming application data to XML" on page 460.

Related tasks:

"Generating mappings from language structures" on page 455
To create XML from application data or vice versa, you create the mappings to describe how CICS will transform the data and XML at run time. You can start from any application data record; for example, you can start with a COMMAREA, VSAM file, temporary storage queue, or a DB2 record.

"Generating mappings from an XML schema" on page 458
To create application data from XML or vice versa that conforms to an existing XML schema, you create the mappings to describe how CICS will transform the data at run time. You can start from an XML schema or a WSDL document.

Querying XML from an application

You can write an application program to query a fragment of XML before transforming it into application data.

About this task

If your application is going to process many different types of XML, you might want to query the XML to determine which XMLTRANSFORM resource to use to transform it to application data. This command might also be useful if your XML contains <xsd:any> elements.

Procedure

1. In your application program, use the **TRANSFORM XMLTODATA** API command to query the XML:

```
EXEC CICS TRANSFORM XMLTODATA
     CHANNEL('MyChannelName')
     XMLCONTAINER('SourceContainerName')
     ELEMNAME(elementName) ELEMNAMELEN(elementNameLength)
```

You must specify the name of the channel and the name of the container that holds the XML. You do not have to specify an XMLTRANSFORM resource to query the XML. The above example queries the name of the first XML element and the length of the XML element. You can also query the type of the first XML element, the length of the type, and the namespace of the type.

2. Optional: If the application requires the namespace of the XML element, provide a data-area in which CICS can write the ELEMNS value.
3. Optional: After querying the XML, you can write application logic to determine which XMLTRANSFORM resource to use to transform the XML to application data.
4. Install the application program in CICS.

Results

CICS reads the specified container and returns the information about the XML element to the application program.

Handling XML by data type

If the XML schema contains global data types, one or more of which are referenced in the XML, you can generate metadata to support these global data types and then parse or transform the XML in an application program.

Before you begin

To generate the correct metadata, you must run DFHSC2LS specifying the `TYPES=ALL` parameter.

About this task

Procedure

1. To transform application data to XML by data type, use the **TRANSFORM DATATOXML** command:

```
EXEC CICS TRANSFORM DATATOXML
     XMLTRANSFORM('MyXmlTransformName')
     CHANNEL('MyChannelName')
     DATCONTAINER('SourceContainerName')
     XMLCONTAINER('TargetContainerName')
     ELEMNAME(elementName) ELEMNAMELEN(elementNameLength)
     ELEMNS(elementNamespace) ELEMNSLEN(elementNamespaceLength)
     TYPENAME(typeName) TYPENAMELEN(typeNameLen)
     TYPENS(typeNamespace) TYPENSLEN(typeNamespaceLen)
```

MyXmlTransformName is the 32-character name of the XMLTRANSFORM resource that specifies the XML binding and schema; *MyChannelName* is the 16-character name of the channel that has the input and output containers; *SourceContainerName* is the 16-character name of the input container that holds the application data and *TargetContainerName* is the 16-character name of the output container that CICS populates with XML. You must also specify the XML element name, the namespace, the data type, and the namespace of the data type for the transformation.

2. To parse the XML by data type, use the **TRANSFORM XMLTODATA** command. The options that you specify on the command depend on whether the XML has an xsi:type attribute.
   - If the XML uses the xsi:type attribute, specify the following command in your application program:

     ```
     EXEC CICS TRANSFORM XMLTODATA
          XMLTRANSFORM('MyXmlTransformName')
          CHANNEL('MyChannelName')
          XMLCONTAINER('SourceContainerName')
          DATCONTAINER('TargetContainerName')
          ELEMNAME(elementName) ELEMNAMELEN(elementNameLength)
          TYPENAME(typeName) TYPENAMELEN(typeNameLength)
     ```

     If the application also requires the namespace of the data type, add the TYPENS option to the API command. CICS returns the data type from the xsi:type attribute in TYPENAME.
   - If the XML does not use the xsi:type attribute, the application program can specify the local name and namespace of the global data type. Use the following command in your application program:

     ```
     EXEC CICS TRANSFORM XMLTODATA
          XMLTRANSFORM('MyXmlTransformName')
          CHANNEL('MyChannelName')
          XMLCONTAINER('SourceContainerName')
          DATCONTAINER('TargetContainerName')
          TYPENAME(typeName) TYPENAMELEN(typeNameLength)
          TYPENS(TypeNamespace) TYPENSLEN(typeNamespaceLen)
     ```

     The application program must specify the type name and namespace but not the name and namespace. This command indicates that the application tells CICS which data type to use. When CICS is generating XML, CICS always adds the xsi:type attribute.
3. Install the application program in CICS.

What to do next

Test that the application program generates and parses the XML as expected.

Handling <xsd:any> data types

If you are working with an XML schema that contains one or more <xsd:any> data types, the XML assistants can map the data type to a pair of CICS containers. You can write an application program to parse the XML in the containers.

Before you begin

You must map the XML schema using DFHSC2LS or DFHWS2LS using a mapping level of 2.1 or higher.

About this task

When CICS transforms the data to XML, it puts the XML associated with the <xsd:any> data type in the first container and the namespace prefix declarations that are in scope in the second container.

Procedure

1. To parse the XML data, use the **TRANSFORM XMLTODATA** command in your application program:

```
EXEC CICS TRANSFORM XMLTODATA
     XMLTRANSFORM('MyXmlTransformName')
     CHANNEL('MyChannelName')
     XMLCONTAINER('SourceContainerName')
     DATCONTAINER('TargetContainerName')
     NSCONTAINER('NamespacesContainerName')
     ELEMNAME(elementName) ELEMNAMELEN(elementNameLength)
```

MyXmlTransformName is the 32-character name of the XMLTRANSFORM resource that specifies the XML binding and schema; *MyChannelName* is the 16-character name of the channel that has the input and output containers; *SourceContainerName* is the 16-character name of the input container that holds the XML and *TargetContainerName* is the 16-character name of the output container that CICS populates with application data; *NamespacesContainerName* is the 16-character name of the container that CICS populates with the namespace prefix declarations. Provide initial values for the ELEMNAME and ELEMNAMELEN options. CICS returns the XML element and its length in the ELEMNAME and ELEMNAMELEN options.

2. Install the application program in CICS.

What to do next

Test that the application program parses the XML correctly.

Related reference:

"Support for <xsd:any> and xsd:anyType" on page 442
DFHWS2LS and DFHSC2LS support the use of <xsd:any> and xsd:anyType in the XML schema. You can use the <xsd:any> XML schema element to describe a section of an XML document with undefined content. xsd:anyType is the base data type from which all simple and complex data types are derived; it has no restrictions or constraints on the data content.

Validating XML transformations

When you use the CICS XML assistant to map application data to XML, you can specify that the transformations that take place at run time are validated to ensure that they conform to the schema that is contained in the XML binding. You can perform validation on the transformation of XML to binary data or binary data to XML.

Before you begin

During development and testing of your CICS application, full validation assists in detecting problems in the XML. However, full validation of the XML carries a substantial overhead and it is inadvisable to validate XML in a fully tested production application.

CICS uses a Java program to validate the XML against the schema. Therefore, you must have Java support enabled in your CICS region to perform validation.

Procedure

1. Set up a JVM server in the CICS region. The XML validator class can run in an OSGi framework or Axis2, but not in a Liberty profile. CICS provides samples to quickly set up a JVM server that uses an OSGi framework.
   a. Install the sample JVM server DFH$JVMS in group DFH$OSGI or create your own JVM server. For more information, see Setting up a JVM server in Configuring.

b. If you created your own JVM server, modify the DFHPIVAL program definition in group DFHPIVAL to reference the name of the JVMSERVER resource. The DFHPIVAL definition is not locked and can be edited. By default, the definition references DFH$JVMS.

2. Ensure that the XML binding and the schema are in the same location on z/OS UNIX. The XMLTRANSFORM resource defines these files to CICS. You can use the **INQUIRE XMLTRANSFORM** command to check the location of each file.
3. Turn validation on for the application. In CICS Explorer, open the XMLTRANSFORM resource and edit the Validation Status field in the list of attributes. Alternatively, you can use CEMT or the SPI.

Results

Check the system log to find out if the XML transformation is valid. Message DFHML0508 indicates that the XML was successfully validated and message DFHML0507 indicates that the validation failed.

What to do next

When you no longer require XML validation for the application, update the XMLTRANSFORM resource to turn it off.

Part 6. Business services and bundles

Creating business services that participate in Service-Oriented Architectures from CICS applications and deploying them in bundles.

© Copyright IBM Corp. 1989, 2013

Chapter 20. Creating business services from CICS applications

Applications that are business services can participate in a service-oriented architecture (SOA). A *business service* is a service that is aligned with business processes and models rather than a technical implementation. You can expose existing and new CICS applications as part of a business service using the support provided in CICS.

You can create two types of service from your CICS applications:

Channel-based services
: These services use the Service Component Architecture (SCA) support to expose applications as service components. The interface for these services is a channel. Channel-based services can be called only by other CICS applications using the **INVOKE SERVICE** API command.

XML-based services
: These services are typically Web service provider or requester applications that use XML to interface with other applications and use a binding to transform the data. You can also describe Web service applications as components using SCA. The interface for these services is XML with a binding to transform the data. XML-based services can be called by other CICS applications using the **INVOKE SERVICE** API command or by an external client.

  An XML-based service can also be an application that uses the **TRANSFORM** API commands to map application data to and from XML. The XML assistant uses a language structure or XML schema to generate the XML binding and also create a bundle.

Both types of service use the pipeline support in CICS.

Service Component Architecture (SCA)

Service Component Architecture (SCA) is a set of specifications that describe a programming model for building applications and systems using a Service-Oriented Architecture (SOA). SCA extends and complements previous approaches to implementing services and builds on open standards such as Web services.

The specifications describe how to create *composite applications*. A composite application is created by combining one or more components that together implement the business logic of the new application. A *component* comprises an application program that implements the business logic and configuration information. An application developer can use the same application program with different configurations to form different components. A component offers a service to other components and in turn consumes functions offered by other services using service-oriented interfaces.

An application developer can assemble components together to create a solution for a particular business requirement. A composite application can contain both new components that are created specifically for the business application and existing components that are reused from other applications.

© Copyright IBM Corp. 1989, 2013

CICS supports the *SCA Assembly Model 1.0* specification, which describes how service components can be assembled to form *composites*. A composite is the unit of deployment in SCA and is described in an XML language called SCDL. Composites can contain components, services, references, property declarations, and the wiring that describes the connections between these elements. Composites can also be used in components with other composites, allowing for a hierarchical construction of composite applications, where high-level services are implemented internally by sets of lower-level services.

Structure of a component

A simple type of component has one service and one reference. A *service* is an addressable interface for the component that can contain one or more operations. A *reference* is a dependency on a service that is provided by another component. The bindings for the component can be defined in both the service and the reference:

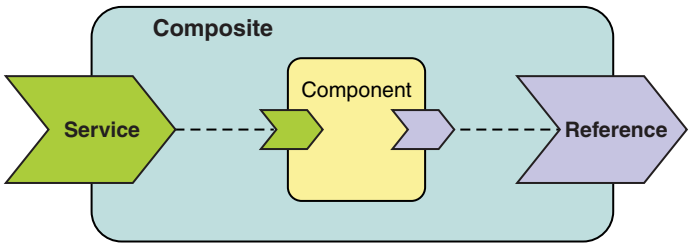
- Component services use bindings to describe the access mechanism that other internal components have to use to call the component.
- Component references use bindings to describe the access mechanism that is used to call other components.

Component services and references are internal and are used only for component-to-component communication. To create an external interface, the component must be deployed inside a composite. A very simple composite has one external service and one external reference:

- Composite services use bindings to describe the access mechanism that external clients must use to call the service.
- Composite references use bindings to describe the access mechanism that is used to call another service.

Composite services and references are not part of the component itself, but are a boundary between the component and other external clients or services.

The following diagram shows a composite that contains a simple component with both internal and external services and references.



SCA composites and wiring

An application developer creates SCA composites by wiring services and references together. A wire is a connector that passes control and data from a component to a target.

You can wire components together in a composite in two ways using Rational Developer for System z:

Promotion

You can wire a composite service by promoting a component service that is defined on one of the components in the composite. Similarly, a composite reference can promote a component reference. You cannot promote a composite service to another composite service or a composite reference to a composite reference. If you add a binding to a service or reference, the binding attributes on a promoted composite service or reference take precedence over attributes on the component service or reference.

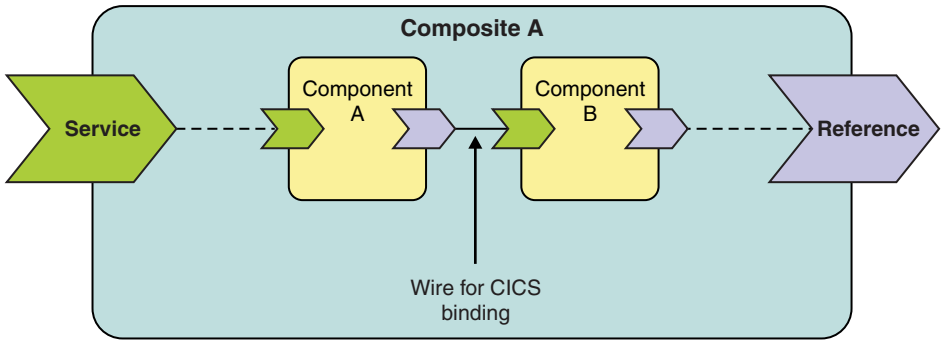
It is not possible to promote an internal service if the external service is using a different type of binding; for example, if an external service has a CICS binding, the internal service cannot have a Web service binding.

Target You can directly wire a component reference to another component service by specifying a target attribute on the reference to connect two components together in a composite. This wiring is internal to the composite. Similarly, you can specify a target attribute on a composite reference to wire two composites together. This wiring is typical in complex composites.

A simple composite

A composite can contain two or more components that are wired together. In the following example, composite A has two components. Component A has a dependency on component B for its service. Each component has a service and a reference. The bindings defined in the reference of component A and the service of component B must be compatible. Composite A encapsulates the two components, hiding the lower-level implementation details from applications that require the service offered by composite A.

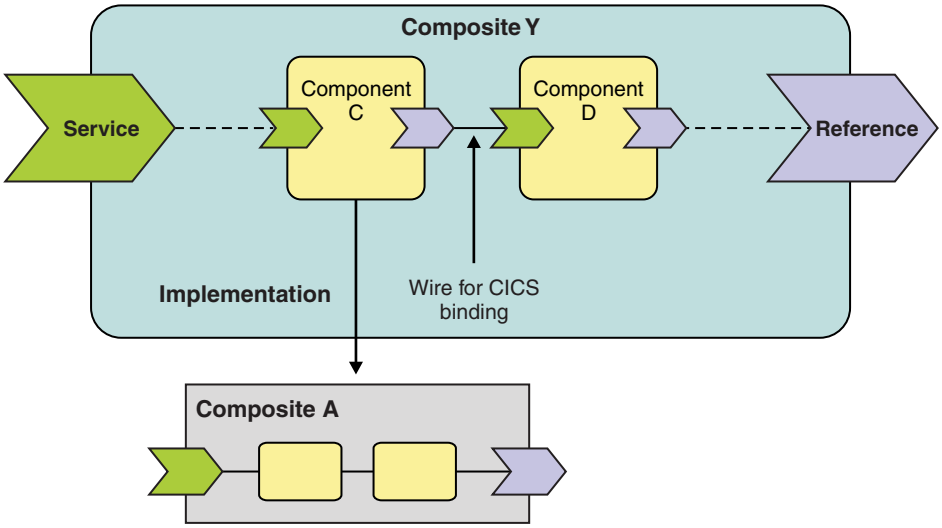
The external service and reference of the composite are formed by promoting the service of component A and the reference of component B. The wire between component A and B is a target, where the bindings on each interface are the same. In this example, the components are using a CICS binding.



A complex composite

The model for encapsulating the implementation details in components is a very flexible way to create hierarchical composite applications. In the following example, composite Y contains two components. However, in this example, the implementation of component C is itself a composite that has two components inside it. Any client that wants to call the service offered by composite Y does not require knowledge of the components that are in composite C, meaning that the underlying implementation details can change without affecting the client.

The external service and reference of composite Y is formed by promoting the service of component C and the reference of component D. The wire between component C and D is a target, where the bindings on each interface are the same. In this example, the components are using a CICS binding.



Best practices for creating and deploying composites

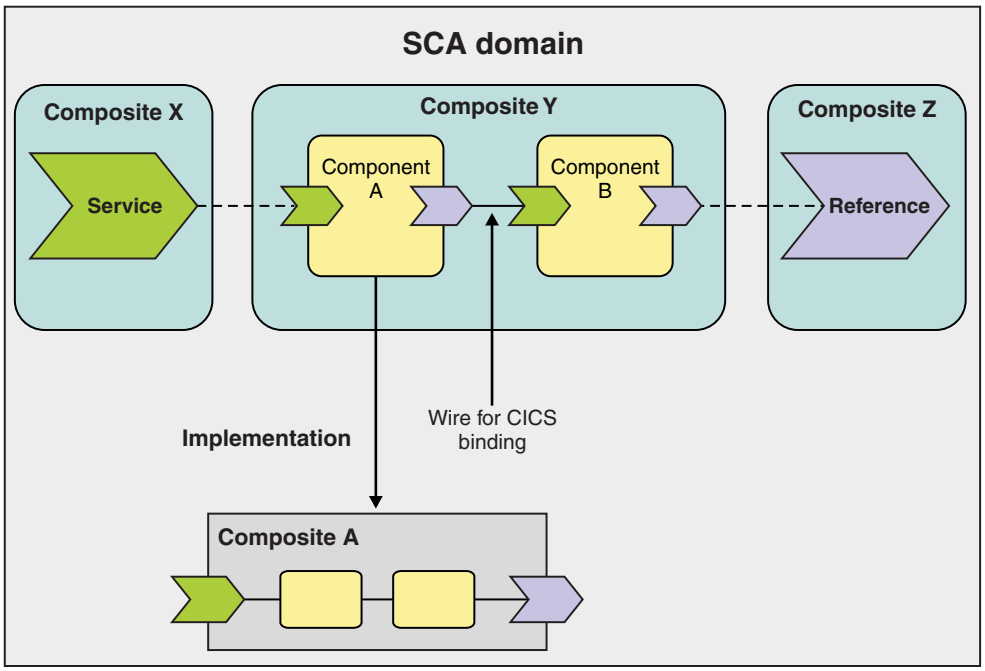
The model for assembling components into composite applications is very flexible. To reuse your application composites in multiple CICS regions, whether you are moving applications through development and test to production or you are cloning applications in your production environment, you are recommended to separate the application logic from the bindings.

Separate implementation logic from the composite service and reference
To more easily reuse the same application logic in different CICS regions, separate the composite service and reference from the implementation logic of the application and deploy them as separate bundles. The composite service and reference can contain information specific to a CICS region, such as a particular set of system resources or a transaction ID. Using this method, you can change or update the composite services and references without having to redeploy the application composite.

Define an SCA domain for the bundle
The bundle is represented in CICS by the BUNDLE resource. Every bundle that is deployed into CICS has the same SCA domain by default, although the value is empty. The BUNDLE resource has an optional attribute called BASESCOPE that the system programmer can use to set an absolute name to represent the SCA domain. If you follow the recommended model of splitting the service and reference from the composite application and deploy them as separate bundles, you can request that the same BASESCOPE value is used on each BUNDLE resource to indicate that the bundles are related.

You can also deploy the same bundle multiple times into the CICS region by specifying different SCA domains for the BASESCOPE attribute. CICS uses the SCA domain and the composite together to identify the service

during runtime processing. The scope of the service is available to the task that is processing the request. It is recommended that the value of the BASESCOPE is a unique URI.



Creating a channel-based service

Channel-based services are CICS applications that are described as components and assembled together using a tool such as Rational Developer for System z. These services are available only to other CICS applications that use the **INVOKE SERVICE** API command and pass binary data in containers on a channel.

About this task

In SCA, the application program that you want to expose as a channel-based service is the implementation of the business logic. The component service describes the interface to the application program in WSDL. A component has one interface, which can contain operations and bindings. For a channel-based service, a CICS binding describes the channel. If you select this binding, Rational Developer for System z adds a `binding.cics` section to the SCDL. A channel-based service has no data mappings, because the application interface expects a channel and binary data in containers.

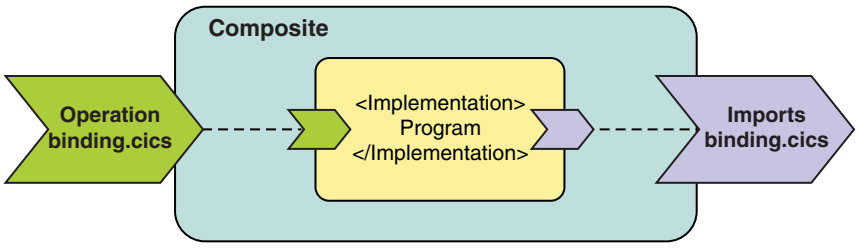
Procedure

1. Create a composite using Rational Developer for System z.
   a. Specify the application program name as the component implementation.
   b. Add the CICS binding to the appropriate service or reference by selecting the `direct` mapping mode in the tooling.
      - If you specify a URI on the CICS binding that is attached to the composite service, this URI provides the external name for the service. In CICS, this URI is a relative path name; for example, if the service is to be

exposed as `http://myhost:port/myService`, the URI in the binding is `myService`, because the host and port are beyond the control of the composite application.

- If you specify a URI on the CICS binding that is attached to the composite reference, the URI is complete for the targeted service; for example `http://myhost:port/myService` or `cics://PROGRAM/prog1?user=user1`.

c. Optional: Define imports on the composite reference. Imports define the dependencies of the component or composite.



For details about the recommended ways to create and deploy composites in CICS, see "Best practices for creating and deploying composites" on page 472.

2. Deploy the composite to CICS as a bundle. Rational Developer for System z generates the bundle manifest and packages the SCDL and other artifacts for you. The manifest describes all the resources and metadata that CICS requires to successfully install a BUNDLE resource; the BUNDLE resource represents the composite in the CICS region. The manifest defines the composite as an SCACOMPOSITE resource type and references the location of the SCDL using a relative path.
3. Create and install the BUNDLE resource. You can optionally set an SCA domain on the BUNDLE resource definition. An SCA domain typically represents a set of services that provide an area of business function. You can install the same bundle using different SCA domains, because CICS identifies the service by combining the SCA domain and the name of the service. For details and examples of how to add an SCA domain, see the *CICS Resource Definition Guide*.

   You must ensure that all prerequisites of the bundle are available in the CICS region for the BUNDLE resource to install successfully.
4. Write an application to call the channel-based service using the **INVOKE SERVICE** API command:

```
EXEC CICS INVOKE SERVICE('servicename')
          CHANNEL(channel)
          OPERATION(operation)
```

   The *servicename* is the external name of the service, the *channel* is the 16-byte name of the channel, and the *operation* is any value. Although the operation is a mandatory option on the command, the value is not used for channel-based services.

Results

When the application calls the channel-based service, CICS resolves the name of the service and issues an **EXEC CICS LINK** command to pass the specified channel and containers to the application program that you defined in the composite. If you specified a specific requester pipeline in the binding, CICS runs the request through that requester pipeline. If no requester pipeline is specified in the binding,

CICS dynamically creates a requester pipeline for the request.

What to do next

You can test and validate that the service works as expected. You can view the BUNDLE resource and its contents using the IBM CICS Explorer. You can also enable and disable the BUNDLE resource to manage all the resources together.

Creating an XML-based service

XML-based services are Web service provider or requester applications that use XML to interface with other applications and use a binding to transform the data. XML-based services are available to CICS applications that use the **INVOKE SERVICE** API command or to business services that are on an external network.

About this task

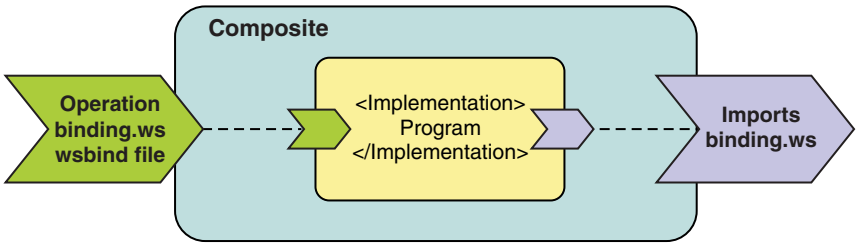
You can either create Web services using the Web services support in CICS or you can use Rational Developer for System z. If you use Rational Developer for System z, you can also create an SCA component from your Web service. The advantages to creating a component from a Web service are as follows:

- You can more easily reuse existing components to rapidly develop new composite applications using Rational Developer for System z.
- You can use SCDL to describe the Web service, moving the configuration information out of the application and into metadata that is easier to change without having to change the application. For example, if you want to run a Web service under different transaction and user IDs, you can change the SCDL without having to regenerate the Web service binding file.

In SCA, the application program that you want to expose as a Web service is the implementation of the business logic. The application program is defined in the <Implementation> element of a component. The component service describes the interface to the application program in WSDL. A component has one interface, which can contain a number of operations and bindings.

Procedure

1. Create a composite using Rational Developer for System z.
   a. Specify the application program name as the component implementation.
   b. If you want the service to be available externally to CICS, add the Web service binding to the appropriate service or reference. If you want the service to be available to CICS applications only, add the CICS binding to the appropriate service or reference and select the `mapped` mapping mode in the tooling. The tooling describes the binding in the SCDL. The SCDL also includes the data mappings that transform the XML to the appropriate high-level language.
   c. Define imports on the composite reference. Imports define the dependencies of the composite that must be satisfied in the CICS region.



For details about the recommended ways to create and deploy composites in CICS, see "Best practices for creating and deploying composites" on page 472.

2. Deploy the composite to CICS as a bundle. Rational Developer for System z generates the bundle manifest and packages the SCDL and other artifacts for you. The manifest describes all the resources and metadata that CICS requires to successfully install a BUNDLE resource; the BUNDLE resource represents the composite in the CICS region. The manifest defines the composite as an SCACOMPOSITE resource type and references the location of the SCDL using a relative path.
3. Create and install the BUNDLE resource. You can optionally set an SCA domain on the BUNDLE resource definition. An SCA domain typically represents a set of services that provide an area of business function. You can install the same bundle using different SCA domains, because CICS identifies the service by combining the SCA domain and the name of the service. For details and examples of how to add an SCA domain, see the *CICS Resource Definition Guide*.

   You must ensure that all prerequisites of the bundle are available in the CICS region for the BUNDLE resource to install successfully. When the BUNDLE resource installs successfully, CICS creates the WEBSERVICE and URIMAP resources for you using the information from the manifest and SCDL. CICS also checks that the resources defined in the Imports section of the Reference are present in the CICS region. The Imports define the prerequisites for the application.

Results

Your Web service is successfully installed in CICS.

What to do next

You can test and validate that the Web service works as expected. You can view the BUNDLE resource and its contents using the IBM CICS Explorer. You can also enable and disable the BUNDLE resource to manage all the resources together.

CICS processing of services

When an application calls a service, for example a web service requester sends a SOAP message or a CICS application uses the **INVOKE SERVICE** API command, CICS uses pipelines to process the request.

CICS processes XML-based services in service requester and service provider pipelines. Each type of pipeline has a transport handler that can send requests over the network. A CICS application can be a web service requester or a web service provider to an external client and use the HTTP, HTTPS, JMS, or WebSphere MQ protocols to send and receive requests.

When the web services are both CICS applications, the service requester can use the **INVOKE SERVICE** command and a URI that begins with `cics://` to optimize the pipeline processing. The data transformations to turn the binary data into XML take place as usual, the pipeline runs through the message handlers, and the pipeline transport handler links to another program or starts a pipeline depending on which type of URI is specified.

For example, you can use the `cics://SERVICE/`*`service`*`?targetServiceUri=`*`targetServiceUri`* URI to run the request through a provider pipeline without the overhead of using the network. CICS uses the *service* and *targetServiceUri* values to resolve the request using a URIMAP.

This URI might be useful when the requester and provider applications are written in different languages, or use different mapping levels, and expect different binary data.

CICS processes calls to channel-based services by using an **EXEC CICS LINK PROGRAM** to pass the channel and containers to the program that is defined in the SCDL for the service. If the calling application also provides a URI that begins with `cics://`, CICS can perform additional processing in a requester pipeline. If the binding does not specify a requester pipeline, CICS creates one dynamically for the request.

URI formats

If the URI begins with `cics://`, the pipeline transport handler can either link to a program, start another requester pipeline, or start a provider pipeline. There are three types of URI:

cics://PROGRAM/*program*
: The pipeline processing of this URI type is similar to the local optimization that takes place for Web service requester and providers that are in the same CICS region. *program* defines the name of the program that the transport handler links to at the end of the requester pipeline processing. The transport handler uses an **EXEC CICS LINK PROGRAM** command to pass the channel and its containers to the specified program. CICS can also pass a COMMAREA to the program if an additional option appears in the URI.

cics://SERVICE/*service***?targetServiceUri=***targetServiceUri*
: The pipeline processing of this URI type optimizes the request by not sending it over the network. However, the request runs through both a requester and provider pipeline. The *service* value defines the name of a service rather than a specific program. The *targetServiceUri* value defines the path of the service, which is resolved by a URIMAP resource. The transport handler uses the URIMAP to send the request to the correct provider pipeline in the same CICS region.

cics://PIPELINE/*pipeline***?targetServiceUri=***targetServiceUri*
: The *pipeline* value defines the name of a requester pipeline and the transport handler puts the value of *targetServiceUri* in DFHWS-URI before starting the specified requester pipeline. This URI type can chain a number of requester pipelines together, so that the request can be processed by different sets of message handlers.

Alternatively, a message handler in the pipeline can override the URI provided by the application to control the pipeline processing.

For details of the parameters that you can specify on these URIs, see the *CICS Web Services Guide*.

Troubleshooting SCA problems

The problems that you might get when implementing SCA composite applications in CICS can occur during the deployment process or at run time, when CICS is processing service requests.

About this task

CICS issues DFHPI or DFHRL prefixed messages to the CICS log when an error occurs. Typically, messages include a user response that provides additional actions that you can take to solve the problem.

Procedure

- The BUNDLE resource installs in a disabled state and you cannot enable it. The mostly likely cause is that one of the dynamically created resources has installed in an UNUSABLE state.
  1. Use the IBM CICS Explorer to view the resources that CICS has created for the bundle.
  2. Determine which resource is in the UNUSABLE state and fix the problem. Any additional DFHPI or DFHRL messages in the log might indicate why the resource installed in this state.
  3. When you have fixed the problem, discard the BUNDLE resource and reinstall it.
- The BUNDLE resource installs in a disabled state and you receive a DFHPI2005 error message. If you receive this message, a binding compatibility issue exists in the composite application.
  1. Edit the composite in Rational Developer for System z to change the binding on the composite service or reference. You must ensure that the bindings on the services are both either CICS bindings or Web service bindings.
  2. Redeploy the bundle to CICS.
  3. Discard the disabled BUNDLE resource and reinstall it.
- An application cannot call the service and CICS issues a DFHPI error message. Check that the URI is correct for the composite service.

Part 7. File control

Understanding and operating the CICS file control facilities.

© Copyright IBM Corp. 1989, 2013

Chapter 21. Understanding file control

CICS data management services have traditionally been known as CICS file control. CICS file control 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.

VSAM data sets: KSDS, ESDS, RRDS

CICS supports access to the following types of data set, Key-sequenced data set (KSDS), Entry-sequenced data set (ESDS) and 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 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 4 GB 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.

© Copyright IBM Corp. 1989, 2013

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 cannot 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.

A standard RBA is an unsigned 32 bit number. The use of a 32 bit RBA means that a standard ESDS cannot contain more than 4 GB of data. However, there is a different ESDS that supports 64 bit **extended relative byte addresses** (**XRBA**s) and which is therefore not subject to the 4 GB limit. This type of ESDS is called an "extended format, extended addressing ESDS data set". For brevity, we will refer to it as an "extended addressing ESDS", or as an "**extended ESDS**". From CICS TS for z/OS, Version 3.2 onwards, CICS supports 64 bit XRBAs and extended ESDS data sets.

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, can accept records of any length up to the maximum for the data set. In a variable RRDS VSAM locates the records with 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 data sets 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 can 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 previously. 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 or extended ESDS) 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 to CICS.

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.

With RLS, CICS regions that share VSAM data sets can reside in one or more MVS images within an MVS parallel sysplex. RLS also provides some benefits when data sets are being shared between CICS regions and batch jobs.

If you open a file in RLS mode, locking takes place at the record level instead of the Control-Interval level, thus reducing the risk of deadlocks.

CICS supports record-level sharing (RLS) access to the following types of VSAM data set:

- Key sequenced data sets (KSDS). Note that if you are using KSDS, you cannot use the relative byte address (RBA) to access files.
- Entry sequenced data sets (ESDS). Note that although RLS access mode is permitted for entry sequenced data sets (ESDS), it is not recommended, as it can have a negative effect on the performance and availability of the data set when you are adding records. (See Performance aspects of VSAM record-level sharing in the *CICS Performance Guide*).
- Relative record data sets (RRDS), for both fixed and variable length records.

Note: If you issue the SET FILE EMPTY command for a file that specifies RLS mode, the request is accepted but is ignored all the time the file is opened in RLS mode. If you close and switch the file to non-RLS mode, the data set is then reset to empty (provided it is defined as reusable on its IDCAMS definition).

Most types of data set are eligible to participate in VSAM record level sharing and most CICS applications can benefit from this mode of access. However, there are some limitations that could affect some applications. The following types of file, data set, or method of access are **not** supported in RLS mode:

- RBA access to a KSDS
- Key-range data sets
- Temporary data sets
- VSAM clusters with the IMBED attribute
- Direct opening of an alternate index
- Opening individual components of a cluster
- Access to catalogs or to VVDS data sets
- CICS-maintained data tables
- Hiperbatch

Identifying VSAM records

You can identify records in data sets by *key*, by *relative byte address (RBA)* or *extended relative byte address (XRBA)*, or by *relative record number (RRN)*.

The RIDFLD (record identification field) option on the CICS file control commands identifies a field containing the record identification appropriate to the access method and the type of file being accessed. For most things you can do to a record (read, add, delete, or start a browse), you identify the record by specifying the RIDFLD option, except when you have read the record for update first. However, there is no RIDFLD for ENDBR, REWRITE, and UNLOCK commands.

- "Key"
- "Relative record number (RRN), relative byte address (RBA) and extended relative byte address (XRBA)" on page 485

Key

If you use a key, you can specify either a complete key or a generic (partial) key. However, if you write a record to a KSDS or write a record by an alternate index path, 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 record number (RRN), relative byte address (RBA) and extended relative byte address (XRBA)

You can use the RRN, RBA, and XRBA options on most commands that access data sets. In effect, they define the format of the record identification field (RIDFLD).

Unless you specify the RRN, RBA, or XRBA, 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).

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.

RBA and XRBA

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 can 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 can change during the transaction as a result of another transaction adding records to, or deleting records from, the same data set.

An RBA is an unsigned 32 bit number. The use of a 32 bit RBA means that a standard ESDS cannot contain more than 4 GB of data. However, there is a different ESDS that supports 64 bit extended relative byte addresses (XRBAs) and which is therefore not subject to the 4 GB limit. We call this type of ESDS an extended ESDS. From CICS TS for z/OS, Version 3.2 onwards, CICS supports 64 bit XRBAs and extended ESDS data sets.

Normally, to access an extended ESDS a program supplies a 64 bit XRBA. You can convert existing programs that use a 32 bit RBA to use a 64 bit XRBA, by replacing

the RBA keyword on the relevant commands with the XRBA keyword and changing the lengths of the areas used for keys. Also in some circumstances it is possible to reuse older programs without change to access an extended ESDS, if they pass 32 bit RBAs but do not use them. “Upgrading to extended addressing for ESDS” explains how to upgrade a standard ESDS to an extended addressing ESDS, and how to upgrade or reuse your existing programs with the new format.

Related information:

“Upgrading to extended addressing for ESDS”
To use an extended entry-sequenced data set (ESDS), you must upgrade the data set and convert existing CICS application programs that use 32-bit relative byte addressing (RBA) to 64-bit extended relative byte addressing (XRBA).

Upgrading to extended addressing for ESDS

To use an extended entry-sequenced data set (ESDS), you must upgrade the data set and convert existing CICS application programs that use 32-bit relative byte addressing (RBA) to 64-bit extended relative byte addressing (XRBA).

Upgrading a standard ESDS to an extended addressing ESDS

Note: Before upgrading a standard ESDS to use extended addressing, if your data set is defined to use forward recovery it is essential to upgrade your forward recovery product to one that can read the new log records written for extended addressing ESDSs. If you use CICS VR, you must use CICS VSAM Recovery for z/OS V4.2 or higher.

To convert an existing standard ESDS to an extended addressing ESDS, you must recreate the data set as follows:

1. If you want to continue to use the contents of the existing data set, take a copy of its contents. You can use the `AMS REPRO` function to do this.
2. Delete the existing data set.
3. Create a new data set. You can base the AMS definition of the new data set on that of the old data set. The only mandatory change is that the **`DATACLAS`** parameter of the new data set definition must name an SMS data class that specifies both extended format and extended addressing.

   How to define SMS data classes is described in the *DFSMS Storage Administration Reference* manual.
4. If required, restore the contents of the data set from the copy taken previously.

Upgrading a program from 32-bit RBA to 64-bit XRBA

To convert an existing program from using 32-bit RBA to 64-bit extended relative byte addressing (XRBA) you must:

1. Replace the RBA keyword with the XRBA keyword, on all the following commands:
   - EXEC CICS READ
   - EXEC CICS READNEXT
   - EXEC CICS READPREV
   - EXEC CICS RESETBR
   - EXEC CICS STARTBR
   - EXEC CICS WRITE
2. Replace all 4-byte areas used for keys with 8-byte areas. If you change "RBA" to "XRBA" but don’t change the length of the key areas:

a. On STARTBR and READ commands, CICS will treat your 4-byte RBAs as being the top half of 8-byte XRBAs. In most cases this produces a huge XRBA number. You can track down this error quite quickly because the program immediately receives a "no record at RBA" response.
b. WRITE commands might produce more subtle errors. The command returns an 8-byte XRBA that overwrites the 4 bytes immediately following the key area.

Using RBA-insensitive programs to access extended ESDS data sets

It is possible to reuse existing 32-bit RBA programs, that do not make use of the RBAs, to access 64-bit extended ESDSs.

For example, there is a common type of application in which records are first written sequentially and later browsed sequentially from the beginning. Although RBAs are passed between CICS and the program, the program makes no use of them. The program reads or writes the next record only. These programs are called “RBA-insensitive”. Other programs, such as those that directly read or update records at named RBAs, are called “RBA-sensitive”.

Existing 32-bit RBA-insensitive programs can access 64-bit extended ESDSs without change. Both RLS and non-RLS modes are supported.

Thirty-two-bit RBA-sensitive programs cannot access 64-bit extended ESDSs, even if the data set contains less than 4 gigabytes of data.

Connecting a back-level AOR to a CICS TS for z/OS, Version 5.1 FOR

In this scenario, old-style 32-bit RBA programs try to access files on a CICS TS for z/OS, Version 5.1 file-owning region (FOR). This access works in either of the following cases:

- The target file in the FOR has not been converted from conventional ESDS to extended addressing ESDS.
- The target file has been converted to extended addressing ESDS but the program is RBA-insensitive.

If the target file has been converted to extended addressing ESDS, a 32-bit RBA-sensitive program running in the AOR cannot access it. The program receives an ILLOGIC response.

Connecting a CICS TS for z/OS, Version 5.1 AOR to a back-level FOR

In this scenario, new-style 64-bit XRBA programs try to access files on a back-level file-owning region.

Because the target region supports only 32-bit RBAs, it does not understand a 64-bit XRBA. The program receives an ILLOGIC response.

Locking of VSAM records in recoverable files

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 sync point request or terminates (at which time a sync point is automatically performed).

The prevention of transaction deadlocks in terms of the record locks acquired whenever records in a recoverable file are modified is explained in "Transaction deadlocks" on page 500. 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 sync point.
- 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 previously are known as update locks, because they are acquired whenever a record is updated (modified). A further type of lock (a delete lock) can 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 can 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 can, 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 enqueuing mechanism is only for updates and deletes and does not prevent a read request being satisfied before the next sync point. 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 that 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 can 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 491).

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 488) 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. The two types are *exclusive* and *shared* locks.

Exclusive locks can be active or retained; shared locks can only be active (see "Active and retained states for locks" on page 491). 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. 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 unrecoverable data sets, are held until the next sync point.

Exclusive locks against records in a unrecoverable 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 can 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 UNLOCKcommand completes, but can 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 can 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 can 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 sync point. 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 118. 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 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 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 that 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 unrecoverable locks are discarded. Only locks that are both exclusive and apply to recoverable resources are eligible to become retained.

BDAM data sets

CICS supports access to keyed and nonkeyed BDAM data sets. BDAM support uses the physical nature of a record on a DASD device.

BDAM data sets consist of unblocked records with the following format:

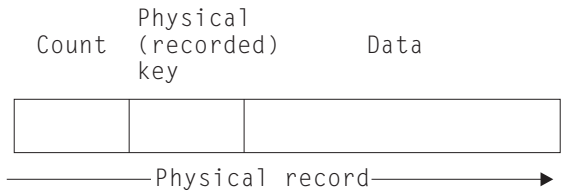


Figure 119. Format of unblocked records in a BDAM data set

Keyed BDAM files have a physical key identifying the BDAM record. The count area contains the physical key length, the physical data length, and the record's data location.

CICS can define a further structure on top of BDAM data sets, introducing the concept of blocked-data sets:

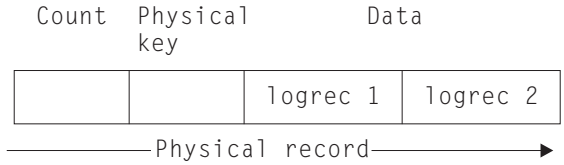


Figure 120. Blocked-data set

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 can 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 21, "Understanding file control," on page 481.

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 RIDFLD (record identification field) option on the CICS file control commands identifies a field containing the record identification appropriate to the access method and the type of file being accessed. For most things you can do to a record (read, add, delete, or start a browse), you identify the record by specifying the RIDFLD option, except when you have read the record for update first. (However, there is no RIDFLD for ENDBR, REWRITE, and UNLOCK commands.)

For BDAM records, the record identification in the RIDFLD option has a subfield for the block reference, the physical key, and the deblocking argument. These subfields, when used, must be in the order given previously.

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 that 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 121 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 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

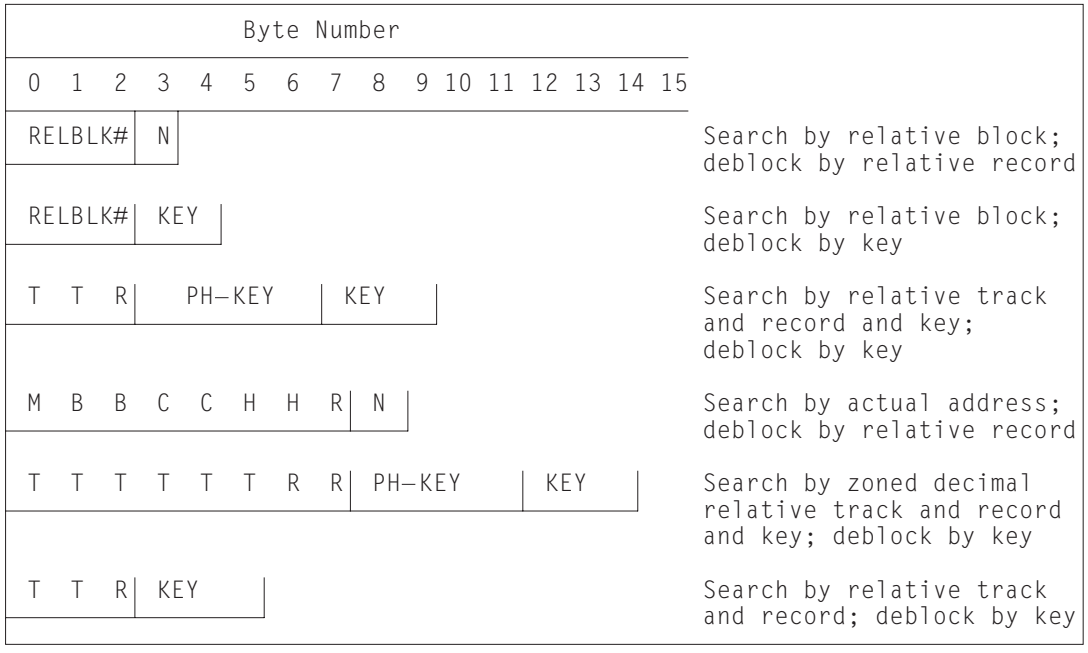


Figure 121. Examples of BDAM record identification

CICS shared data tables

CICS 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 16 MB 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 Shared data tables overview in Shared Data Tables.

A table is defined using the FILE resource. 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 16 MB 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.

  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 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. A table defined as recoverable participates in dynamic transaction backout but is not recovered at restart.

Coupling facility data tables

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 useful for informal shared data. Informal shared data is characterized 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 typically 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 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 can 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:

Unrecoverable
For updates to unrecoverable CFDTs, locks do not last until sync point (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 indoubt and backout failures: if a unit of work fails when backing out an update to the CFDT, or if it fails indoubt during sync point 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

The CICS techniques that can be used for sharing data are compared, in tabular form, to show you when to consider using a coupling facility data table.

Table 29. Techniques for sharing scratchpad data

Constraints and factors	Single Region	Single MVS	Sysplex
Technique no longer recommended (too restrictive)	TWA	—	—

Table 29. Techniques for sharing scratchpad data (continued)

Constraints and factors	Single Region	Single MVS	Sysplex
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 29 on page 497, 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 can be passed from a task in one region to a task in another. '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 30. 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 30, 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 most cases, with a few instances where data tables provide a more appropriate solution for handling sequenced data.

Table 31. Techniques for sharing control records

Constraints and factors	Single Region	Single MVS	Sysplex
Technique no longer recommended	CWA	MVS CSA	—

Table 31. Techniques for sharing control records (continued)

Constraints and factors	Single Region	Single MVS	Sysplex
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 31 on page 498, 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 can 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 consider the named counter function, which is also a sysplex-wide facility. See Chapter 41, "Named counter servers," on page 637 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 32. 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 32, 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 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 indoubt failures,

Transaction deadlocks

Design your applications to avoid transaction deadlocks. A transaction needs exclusive control of resources while executing file control commands.

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.

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 Locks in the *CICS Recovery and Restart Guide*.)

In releases before 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 sync point request. For more information, see "Syncpointing" on page 348.

Whether a deadlock occurs depends on the relative timing of the acquisition and release of the resources by different concurrent transactions. Application programs can 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, 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 sync point.

- 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 like 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

Use the following rules to avoid deadlocks.

- 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 can 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 can 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 with ENDBR commands (releasing the VSAM lock) before issuing a READ UPDATE, WRITE, or DELETE with RIDFLD command, to the file.

File control operations

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.

Using CICS commands to read records

There are three ways for application programs to read records: direct reading using the READ command, sequential reading, and skip-sequential reading. All these methods for reading records can be used on both VSAM and BDAM data sets.

About this task

A file can be defined in the file definition as containing either fixed-length or variable-length records. Define fixed-length records 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 is typically 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)

Records in a file can be read using the READ command with the RIDFLD (record identification field) option to specify a record. Further options are available which describe the content and length of the record identification field.

About this task

When you use the READ command:

Procedure

- Use the RIDFLD (record identification field) option to specify the record you want, and add further options to describe the content of the record identification field. The exact method of identifying the record depends on the type of data set:
  1. For a KSDS, you can identify the record you want uniquely by specifying its full key, or you can specify a generic key and retrieve the first (lowest key) record whose key meets those requirements.
     - The GENERIC option indicates that you require a match on only a part of the key. When you specify the GENERIC option, you must also 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 (greater than or equal to) 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.

     If a KSDS has an alternate index and an alternate index path, you can retrieve a record in the file by specifying the alternate key that you set up in the alternate index (see step 3).
  2. For a standard (non-extended) ESDS, you can identify the record by specifying its relative byte address (RBA).

     Add the RBA option to inform CICS that an RBA is being used. 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 for a KSDS.
  3. For an extended ESDS, you can identify the record by specifying its extended relative byte address (XRBA).

Add the XRBA option to inform CICS that an XRBA is being used. Because the XRBA of a record in an ESDS cannot change, your application program can keep track of the values of the XRBAs corresponding to the records it wants to access. An XRBA must always point to the beginning of a record. There is no equivalent to the GENERIC or GTEQ options that you can use for a KSDS.

4. For a KSDS or a standard ESDS with an alternate index (an extended ESDS cannot have an alternate index), you can retrieve a record in the file by specifying the alternate key that you set up in the alternate index. When you use an alternate key:
   - The GENERIC option and the GTEQ option can be used for both KSDS and ESDS records, in the same way as for a read from a KSDS using the primary key.
   - If the alternate key 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.
   - If no matching record is found, the record identified is the one with the next higher alternate key.
5. For an RRDS, identify the record by specifying its relative record number (RRN). Add the RRN option to inform CICS that an RRN is being used. 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 for a KSDS, and you cannot use an alternate key.

- Specify the KEYLENGTH option if required, to state the length of the key which you specified in the RIDFLD option.
  1. If you specify an RBA or RRN for the record identification field, and specify the RBA or RRN option, the KEYLENGTH option **is not** required.
  2. If you specify the GENERIC option, the KEYLENGTH option **is** required. Specify the length of the generic key, which must be less than the key length specified in the VSAM definition.
  3. If you are reading a remote file and specify the SYSID option, the key length can be identified by any of the following means:
     - Specified in the file definition.
     - Specified by the application program using the KEYLENGTH option.
     - Defaulted to 4, if the key is longer than 4 characters, and the key length is not specified in the file definition or by the application program.

     This is different from the situation for the WRITE command, where the KEYLENGTH option is required for remote files, unless you have used an RBA or RRN.
  4. In other situations, the KEYLENGTH option may be specified or omitted. If you specify a KEYLENGTH which is different from the length defined for the data set and the operation is not generic, the INVREQ condition occurs.
- Use the INTO or SET option to state whether the record is to be read into an area of main storage provided by your application program (READ INTO), or into an area of main storage 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.
- Make sure that the LENGTH parameter is set if required:

1. If you have used the SET option, you do not have to specify the LENGTH option. However, you may specify it if you want to check whether the record length agrees with that originally defined to VSAM. The data area specified is set to the actual record length after the record has been retrieved.
2. For fixed-length records, you do not have to specify the LENGTH option, but it is advisable to do so. If you specify the option, CICS checks that the record is not too long for the available data area. The length you specify should exactly match the record length specified at the time the file was created.
3. If you have used the INTO option for reading variable-length records, a length parameter must be provided. However, the LENGTH option does not always need to be specified explicitly on the command:
   - For a file on a remote system, the LENGTH parameter need not be set here, but it must be set in the file resource definition.
   - If you are using assembler language or PL/I, instead of specifying LENGTH explicitly, you can allow it to be defaulted using the length attribute reference in assembler language, or STG and CSTG in PL/I. LENGTH must be specified explicitly in C.

   If you do specify the LENGTH option explicitly when reading a variable-length file, always specify the longest record your application program accepts, which should correspond to the value defined as the maximum record size when the data set was created. If the record exceeds the length specified, the LENGERR condition occurs, and the record is then truncated to that length. After the record has been retrieved, the data area specified in it is set to the actual record length (before any truncation occurs).

- If the file is open in RLS mode, use the UNCOMMITTED, CONSISTENT, REPEATABLE and NOSUSPEND options to control read integrity.
  1. If you specify the UNCOMMITTED option, there is no read integrity and shared locks are not used for read requests. “RLS Record level locking” on page 489 explains 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.
  2. If you specify the CONSISTENT option, 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. “Syncpointing” on page 348 explains UOWs and syncpoints.
  3. If you specify the REPEATABLE option, processing of the read request is the same as for consistent read requests, except that 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.
  4. If you specify either CONSISTENT or REPEATABLE, you can also specify the NOSUSPEND option to ensure that the request does not wait if the record is locked by VSAM with an active lock. “Active and retained states for locks” on page 491 explains active locks.

  If you do not specify any of the options, the value from the file resource definition is used.

  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.

Sequential reading (browsing)

Use the **STARTBR**, **READNEXT**, **READPREV**, and **RESETBR** commands to browse records in a file. You can browse forwards or backwards, and change the position or characteristics of the browse while the browse is in progress. Use the **ENDBR**, **SYNCPOINT** and **SYNCPOINT ROLLBACK** commands to end the browse.

About this task

When browsing records, you generally identify and read the records in the same way as for a direct read using the **READ** command (see "Direct reading (using READ command)" on page 503). You specify the record identification field (RIDFLD option), and select a destination for the read records. If the file is open in RLS mode, you can use the UNCOMMITTED, CONSISTENT, REPEATABLE and NOSUSPEND options to control read integrity. Some additional special cases for browsing are noted in this topic.

The same types of key are used for browsing as for a direct read:

- For a KSDS, a full key, a generic key, or an alternate key can be used.
- For a standard (non-extended) ESDS, an RBA or an alternate key can be used.
- For an extended ESDS, an extended RBA (XRBA) can be used.

  Note: In some circumstances, a 32-bit RBA can be used to access an extended (64-bit) ESDS: see "Relative record number (RRN), relative byte address (RBA) and extended relative byte address (XRBA)" on page 485.
- For an RRDS, an RRN is used.

If you use an alternate key for browsing, 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.

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.

The instructions in this topic cover general principles for browsing, and specific information for the different types of VSAM data set. For specific information about browsing BDAM data sets, see "Browsing records from BDAM data sets" on page 509.

Procedure

- To start a browse, use the **STARTBR** command. The **STARTBR** command only identifies the starting position for the browse; it does not retrieve a record. Identify a particular record in the same way as for a direct read, and use the RIDFLD option to specify the record identification. See "Direct reading (using READ command)" on page 503, and also note the following considerations:

1. To position the browse at the start of the file, for a KSDS or ESDS, specify a RIDFLD of hexadecimal zeros. For a standard ESDS, also specify the RBA option. For an extended ESDS, specify the XRBA option.
2. To position the browse at the start of the file, for an RRDS, specify an RRN of 1 using the RIDFLD option, and also the RRN option.
3. For a KSDS only, as an alternative method to position the browse at the start of the file, you may specify the options GENERIC, GTEQ, and KEYLENGTH(0). 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. A browsing command with the option KEYLENGTH(0) is always treated as if KEYLENGTH(1) and a partial key of one byte of binary zeros have been specified.
4. To position the browse at the last record in the file, ready for a backward browse, specify a RIDFLD of X'FF' characters. This applies to starting a browse of a KSDS, ESDS, or RRDS. For a standard ESDS, specify the RBA option. For an extended ESDS, specify the XRBA option. For an RRDS, specify the RRN option.
5. For starting a browse of a KSDS, you may specify a generic key using the RIDFLD option. However, if you use a generic key, you can only browse forwards through the file, not backwards.
6. For starting a browse of a KSDS, you may use the options EQUAL (key equal to) and GTEQ (key greater than or equal to), and they have the same meaning as on the **READ** command. However, option GTEQ is the default for the STARTBR command, whereas EQUAL is the default for the **READ** command. The **STARTBR** command assumes that you want to position at the key specified or the first key greater if that key does not exist.
7. For starting a browse of an RRDS, the GTEQ (key greater than or equal to) option is the default on a STARTBR command. When this option is in effect, if the specified RRN does not exist, a pointer is set to the first record having a greater key. Note that the GTEQ option has no effect on a direct **READ** command for a RRDS, and if a direct **READ** command specifies an RRN that does not exist, the NOTFND condition is returned.

- To browse through the records, use the **READNEXT** command. The **READNEXT** command reads records sequentially from the starting point specified by the **STARTBR** command. It operates in a similar way to a direct read. See "Direct reading (using READ command)" on page 503, and also note the following considerations:
  1. Include the RIDFLD option to give CICS a way to return the identifier of each record retrieved, but do not set the field, unless you want to reposition the browse. On completion of each **READNEXT** command, CICS returns the full key of the record it retrieved. Be sure to provide a field as long as the full key, even if you use a **STARTBR** command with a shorter generic key.
  2. You only need the KEYLENGTH option if you are providing a key using the RIDFLD option, to reposition the browse. Otherwise, the current key length is used, as set on the **STARTBR** command or during the last repositioning of the browse.
  3. As for a direct read, you can read the record 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.
- To browse backwards in the file, use **READPREV** commands instead of **READNEXT** commands. The **READPREV** command is like the **READNEXT** command, except that

the records are read sequentially **backward** from the current position. You can switch from one direction to the other at any time. As you switch from one direction to the other, you retrieve the same record twice, unless you reposition.

Note: Remember that for a KSDS, if you used a generic key on the **STARTBR** command, you cannot use the **READPREV** command. An INVREQ condition is returned if you try to use the command in this case.

- To change the current browse position when the browse has started (reposition the browse), use either the **RESETBR** command, or the **READNEXT** command, or the **READPREV** command.
  1. For VSAM, you can reposition the browse by setting a value in RIDFLD when you issue the next **READNEXT** or **READPREV** command.
     - When you set RIDFLD to reposition a browse, the record identifier must be in the same form as on the previous **STARTBR** or **RESETBR** command (key, RBA, XRBA, or RRN). You can use a key, or use hexadecimal zeroes to indicate the beginning of the file, or for a KSDS, specify the options GENERIC, GTEQ, and KEYLENGTH(0) to indicate the beginning of the file. If you use the KEYLENGTH(0) option, note that 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. (X'FF' characters cannot be used for repositioning with the **READNEXT** or **READPREV** command.)
     - 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 the RIDFLD option.

     If you also want to change characteristics of the browse, use the **RESETBR** command instead.
  2. The **RESETBR** command can specify a new browse position in the same ways as the **STARTBR** command. You can:
     - Identify a specific record.
     - Use a key of hexadecimal zeros to indicate the beginning of the file.
     - Use a key of X'FF' characters to indicate the end of the file.
     - For a KSDS, use the options GENERIC, GTEQ, and KEYLENGTH(0) to indicate the beginning of the file. If you use the KEYLENGTH(0) option, note that 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.
- To change the characteristics of the browse (for example, to search with a generic key instead of an exact match), use the **RESETBR** command. You can use the command to:
  - Change the form of the key from key to RBA.
  - Switch between generic and full keys, or between “equal to” and “greater than or equal to” searches.

  Under certain conditions, CICS uses VSAM skip-sequential processing when you change the key, as explained in “Skip-sequential processing” on page 510.
- To end the browse, use the **ENDBR** command. There is no RIDFLD for this command. Trying to browse past the last record in a file raises the ENDFILE condition. 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. You can also end the browse by using the

SYNCPOINT and **SYNCPOINT ROLLBACK** commands. If an implicit sync point occurs at the end of a task, the browse also ends.

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 that 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 command, put the TTR (assuming that is the addressing method) of the first block in the record identification field. After the first 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.

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).

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 wanted 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 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 can incur a long index scan, you might want 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 the same way as for a direct read.

About this task

In a KSDS or ESDS, the record can (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 511.)

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. The REWRITE and UNLOCK commands do not use the RIDFLD option.

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 513). For files opened in non-RLS mode, the application program must end the browse, read the wanted record with a READ UPDATE command, and perform the update. Failure to end the browse before issuing the READ UPDATE command can cause a deadlock.

The record to be updated can (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 can be altered. If the file definition allows variable-length records, the length of the record can be changed.

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 typically (but not necessarily) the same as the corresponding INTO area on the READ UPDATE command.

The length of records in an ESDS, a fixed-length RRDS, or a fixed-length KSDS must not be changed on update. However, the length of the record can be changed when rewriting to a KSDS with variable-length records.

For a file defined to CICS as containing fixed-length records, the length of record being rewritten ***must equal the original length***. However, when writing to a file with fixed-length records, you need not specify 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.

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.

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 513).

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.

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 queuing 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 might not work. It might 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.

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. Specifying HANDLE CONDITION(QBUSY) causes NOSUSPEND to be assumed on subsequent transient data commands even when it is not explicitly specified.

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.

Note: When a blocked BDAM 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.

Using CICS commands to delete records

Use the DELETE command to delete a record or group of records from a file. You may retrieve the record for update first. For a file opened in RLS mode, you may delete a record during a browse.

About this task

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.

The NOSUSPEND option discussed in "Direct reading (using READ command)" on page 503, applies to the CICS browse commands when you are using them to update a file.

Procedure

- Records can never be deleted from an ESDS.
- To delete a single record in a KSDS or RRDS, use one of these three methods:
  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)."
- To delete groups of records in a KSDS or RRDS, use a generic key with the DELETE command. Instead of deleting a single record, all the records in the file whose keys match the generic key are deleted with the single command. If access is by way of an alternate index path, the records deleted are all those whose alternate keys match the generic key. However, this method 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.

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.

About this task

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 is usable. (For more explanation about the TOKEN option, see "The TOKEN option" on page 511.)

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 sync point 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 489.
- 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 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 is not necessary to use UNLOCK in an application program that decides not to update a particular record.

Using CICS commands to add records

Add new records to a file in a VSAM data set using the WRITE command. New records must always be written from an area provided by the application program.

About this task

These instructions cover general principles for writing records, and specific information for the different types of VSAM data set. For specific information about adding records to BDAM data sets, see "Adding records to BDAM data sets" on page 516.

When you use the WRITE command:

Procedure

- To add a record to a KSDS, use the RIDFLD (record identification field) option to specify the base key of the record. 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.
- To add a record to an ESDS base data set, the record must be 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.
- To add a record to a KSDS or standard ESDS by way of an alternate index path, use the RIDFLD option to specify the alternate key. For a KSDS, the record is inserted into the data set in the position determined by the base key. For an ESDS, the record is placed at the end of the data set.

Note: Extended ESDS data sets do not support alternate indexes.

- To add a record to an RRDS, use the RIDFLD option to specify the relative record number. The record is stored in the file in the position corresponding to the RRN.
- Specify the KEYLENGTH option if required, to state the length of the key which you specified in the RIDFLD option.
  1. If you specify an RBA, XRBA, or RRN for the record identification field, and specify the RBA, XRBA, or RRN option, the KEYLENGTH option **is not** required.
  2. In all other situations, the KEYLENGTH option **is** required. If you specify a KEYLENGTH which is different from the length defined for the data set, the INVREQ condition occurs.
- Use the FROM option to specify 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.
- Specify the LENGTH option if required.
  1. When writing to a file with fixed-length records, the LENGTH option is not required. CICS uses the length specified in the cluster definition as the length of the record to be written. If you do specify the LENGTH option, its value is checked against the defined value, and you get a LENGERR condition if the values do not match.
  2. When writing to a file with variable-length records, always include the LENGTH option. 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.
- If the file is open in RLS mode, and there is a possibility that the record might already be locked, you can specify the NOSUSPEND option. The NOSUSPEND option ensures that the request does not wait if the record is locked by VSAM with an active lock. "Active and retained states for locks" on page 491 explains active locks.
- If you have more than one record to add to a KSDS, ESDS, or path, and the keys in successive requests are in ascending order, using the MASSINSERT option improves performance. (The performance improvement is only obtained if the keys are in ascending order.)
  1. Specify the MASSINSERT option on each WRITE command in the sequence.
  2. When the MASSINSERT operation is complete and all records have been written, issue an UNLOCK command to ensure 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 might get unpredictable results, possibly including a deadlock. If you do not issue the UNLOCK command, the MASSINSERT operation is completed when a sync point 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 in "Efficient data set operations" on page 517 for more information about MASSINSERT.

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 timeouts 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.

Adding records to BDAM data sets

Records added to a BDAM data set can be undefined or variable-length (keyed or nonkeyed), keyed fixed-length, nonkeyed fixed-length, or variable-length blocked.

About this task

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 can 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 can 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 2 bytes specify the length of the record (including the 4 byte RDF); the other 2 bytes consist of zeros.

Efficient data set operations

The efficiency of database and data set operations is an important factor in the performance of any CICS system. Use the following information to maximize response times using VSAM and BDAM data sets.

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 typical in these situations, any improvements you make in one area can 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 483, 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 sync point. 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 unrecoverable 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 FILE definition attributes in 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 1 shows which commands hold position and when the hold is released.

Table 33. 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 (STARTBR 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 many output screens, which can monopolize system resources. You can allow other tasks to get control by issuing DELAY or SUSPEND commands.

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 779.

Part 8. Terminal control

The CICS application programming interface contains two sets of commands for communicating with terminals, Terminal control commands and 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 various 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 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 Basic mapping support in Developing applications.

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 section describes:

- Chapter 22, “Terminal access method support,” on page 521
- Chapter 23, “Terminal control commands,” on page 523
- Chapter 24, “Using data transmission commands,” on page 527
- Chapter 25, “Device control commands,” on page 529
- Chapter 26, “Terminal device support,” on page 531
- Chapter 27, “Finding out about your terminal,” on page 535
- Chapter 28, “Using SNA,” on page 539
- Chapter 29, “Using sequential terminal support,” on page 543
- Chapter 30, “Using batch data interchange,” on page 545

© Copyright IBM Corp. 1989, 2013

Chapter 22. Terminal access method support

CICS Transaction Server for z/OS, Version 5 Release 1 supports terminals directly through interfaces to the following access methods, z/OS Communications Server, Basic Graphics Access Method (BGAM) for graphics terminals using GDDM® and 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*.

© Copyright IBM Corp. 1989, 2013

Chapter 23. Terminal control commands

Terminal control commands are used for basic data transmission, for control functions, and for retrieving information about your terminal. Terminal control also provides special device group commands.

The commands described in this section 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 z/OS Communications Server.

Note: This section 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 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 by the z/OS Communications Server for SNA. 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 the Communications Server, 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 225).

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 by issuing a RECEIVE, without a preceding INVITE; INVITE optimizes transmissions.

© Copyright IBM Corp. 1989, 2013

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 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 Communications Server or 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.

But 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

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 with bracket protocol” on page 541), 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 typically 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 typically does on a SEND before a RECEIVE, or on the last SEND in the task. This means that 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 the z/OS Communications Server, violating this protocol causes the task to abend (code ATCV) unless read-ahead queuing is in force. Read-ahead queuing 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 queuing is applied at the transaction level (it is specified in the RAQ option of the PROFILE under which the transaction runs). Read-ahead queuing applies only to LU type 4 devices, and was originally provided for compatibility reasons, to allow a transaction to support both BTAM-connected and z/OS Communications Server-connected terminals in the same way. As BTAM is no longer supported, read-ahead queuing should no longer be used.

Sequential terminals differ from others in send/receive rules. Because the input is a pre-prepared file, CICS 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

The z/OS Communications Server 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 Communications Server, 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 z/OS Communications Server, the bind image must indicate support for it as well, or the z/OS Communications Server ignores the interrupts.

Terminal waits

When you use the WAIT option on a SEND command, CICS does not return control to your task until the output operation is complete.

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.

Chapter 24. Using data transmission commands

Three commands transmit data to and from the terminal or logical unit that is the principal facility of your task.

The commands are:

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 36 on page 533.

The SEND, RECEIVE, and CONVERSE commands are fully described in CICS command summary in Reference -> Application development. They are broken down by device group, because the options for different devices and access methods vary considerably. Chapter 26, "Terminal device support," on page 531 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, the application receives the data that was sent by the terminal. 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.

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 539.

© Copyright IBM Corp. 1989, 2013

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 540 for more on this subject.)

NOTRUCATE option

An 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, issue RECEIVE commnads 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 fulfill 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 in “Printing display screens” on page 673 for more information about the “print” key.)

Chapter 25. 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 following table, along with a brief description of their function. Not all these commands apply to all terminals, and for some, different forms apply to different terminals. See Chapter 26, "Terminal device support," on page 531.

The terminal in the following table is always the principal facility of the task issuing the command, except where explicitly stated otherwise. It can be a logical unit of a type not ordinarily considered a terminal.

Table 34. Control commands for terminals and logical units

Command	Action
FREE	Releases the terminal from the task, so that the terminal can be used in another task before the current one ends.
ISSUE COPY	Copies the buffer contents of the terminal named in the TERMID 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 z/OS Communications Server application named in the LUNAME option, at the end of the issuing task.
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 data flow command.
WAIT TERMINAL	Suspends the issuing task until the previous terminal operation has completed.

© Copyright IBM Corp. 1989, 2013

Chapter 26. 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 the type of terminal, to determine the options that apply and the exception conditions that can occur. For some commands, you also need to know what access method is in use. The following tables show 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 Chapter 27, "Finding out about your terminal," on page 535.

To use the tables, look up the terminal type that your program must support in the first column of Table 35. Use the value in the second column to find the corresponding command group in the first column of Table 36 on page 533. The second column of this table shows the access method, and the third shows the commands you can use. The commands themselves are described in full in CICS command summary in Reference -> Application development. Where there is more than one version of a command, the table shows which one to use. This information appears in parentheses after the command.

Table 35. Devices supported by CICS

Device	Use commands for
2260, 2265	2260
3101 (supported as TWX 33/35)	3767
3230	3767
3270 displays, 3270 printers	LU type 2/3
3270 displays, 3270 printers (non-SNA)	3270 logical
3270 displays, 3270 printers	3270 display
SCS printers	SCS
3600 Pipeline mode	3600 pipeline
3601	3600-3601
3614	3600-3614
3630, attached as 3600 (3631, 3632, 3633, 3643, 3604)	Use 3600 entry
3641, 3644, 3646, 3647 (attached as 3767)	3767
3643 (attached as LU type 2)	LU type 2/3
3642, 3645 (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

© Copyright IBM Corp. 1989, 2013

Table 35. Devices supported by CICS (continued)

Device	Use commands for
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	3767
3770 Interactive LU	3767
3770 Full function LU	3790 full function or inquiry
3770 Batch LU (3771, 3773, 3774)	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, 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	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), supported as 3770	Use 3770 entry
System/34 (5340), supported as 3770	Use 3770 entry
System/34 (5340)	System/3
System/36 (supported as System/34)	Use System/34 entry
System/38 (5381), attached as 3770	Use 3770 entry
System/38 (5381), attached as APPC	APPC
TWX 33/35 NTO	3767
WTTY NTO	3767

Table 36. Terminal control commands by device type

Device group name	Access methods	Commands applicable
2260	non-z/OS Communications Server	RECEIVE (2260), SEND (2260), CONVERSE (2260), ISSUE DISCONNECT (default), ISSUE RESET
3270 display	non-z/OS Communications Server	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)	z/OS Communications Server	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)	z/OS Communications Server	RECEIVE (3270 logical), SEND (3270 logical), CONVERSE (3270 logical), ISSUE COPY (3270 logical), ISSUE DISCONNECT (default), ISSUE ERASEAUP, ISSUE PASS, ISSUE PRINT
SCS	z/OS Communications Server	SEND (SCS), CONVERSE (SCS), ISSUE DISCONNECT (default), ISSUE PASS
3600 pipeline	z/OS Communications Server	RECEIVE (3600 pipeline), SEND (3600 pipeline), ISSUE DISCONNECT (default), ISSUE PASS
3600-3601	z/OS Communications Server	RECEIVE (3600-3601), SEND (3600-3601), CONVERSE (3600-3601), ISSUE DISCONNECT (default), ISSUE PASS, WAIT SIGNAL
3600-3614	z/OS Communications Server	RECEIVE (3600-3614), SEND (3600-3614), CONVERSE (3600-3614), ISSUE DISCONNECT (default), ISSUE PASS
3650 interpreter	z/OS Communications Server	RECEIVE (3650), SEND (3650 interpreter), CONVERSE (3650 interpreter), ISSUE DISCONNECT (default), ISSUE EODS, ISSUE LOAD, ISSUE PASS
3650-3270	z/OS Communications Server	RECEIVE (3650), SEND (3650-3270), CONVERSE (3650-3270), ISSUE DISCONNECT (default), ISSUE ERASEAUP, ISSUE PASS, ISSUE PRINT
3650-3653	z/OS Communications Server	RECEIVE (3650), SEND (3650-3653), CONVERSE (3650-3653), ISSUE DISCONNECT (default), ISSUE PASS
3650-3680	z/OS Communications Server	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	z/OS Communications Server	RECEIVE (3767), SEND (3767), CONVERSE (3767), ISSUE DISCONNECT (default), ISSUE PASS, WAIT SIGNAL
3770	z/OS Communications Server	RECEIVE (3770), SEND (3770), CONVERSE (3770), ISSUE DISCONNECT (default), ISSUE PASS, WAIT SIGNAL
3790 full function or inquiry	z/OS Communications Server	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

Table 36. Terminal control commands by device type (continued)

Device group name	Access methods	Commands applicable
3790 3270-display	z/OS Communications Server	RECEIVE (3790 3270-display), SEND (3790 3270-display), CONVERSE (3790 3270-display), ISSUE DISCONNECT (default), ISSUE ERASEAUP, ISSUE PASS, ISSUE PRINT
3790 3270-printer	z/OS Communications Server	SEND (3790 3270-printer), ISSUE DISCONNECT (default), ISSUE ERASEAUP, ISSUE PASS
3790 SCS	z/OS Communications Server	SEND (3790 SCS), ISSUE DISCONNECT (default), ISSUE PASS
LU type 4	z/OS Communications Server	RECEIVE (LU type 4), SEND (LU type 4), CONVERSE (LU type 4), ISSUE DISCONNECT (default), ISSUE PASS, WAIT SIGNAL
Outboard controllers (batch data interchange)	z/OS Communications Server	ISSUE ABORT, ISSUE ADD, ISSUE END, ISSUE ERASE, ISSUE NOTE, ISSUE QUERY, ISSUE RECEIVE, ISSUE REPLACE, ISSUE SEND, ISSUE WAIT
All others	z/OS Communications Server	RECEIVE (Communications Server for SNA default), SEND (Communications Server for SNA default), CONVERSE (Communications Server for SNA default), ISSUE PASS
All others	non-z/OS Communications Server	RECEIVE (non-z/OS Communications Server default), SEND (non-z/OS Communications Server default), CONVERSE (non-z/OS Communications Server default)

Chapter 27. 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 37 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*. The "terminal" cited in column 2 of the table is always the principal facility of the task.

Table 37. 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 SNA network
NUMTAB	Number of tabs required to position the print element in the correct passbook area (for 2980s only)

© Copyright IBM Corp. 1989, 2013

Table 37. ASSIGN command options for terminals (continued)

ASSIGN option	Information returned
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
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 INQUIRE TERMINAL in the *CICS System Programming Reference*.

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 38 on page 537. (Other fields relate only to LU type 6.1, APPC and MRO operations; see CICS command summary in Reference -> Application development 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 38. EIB fields that apply to terminal control commands

Field	Contents
EIBAID	The attention identifier (AID) from the last input operation (3270s only, see the AID in "Input from a 3270 terminal" on page 562).
EIBATT	Whether the input contains attach header data (an attach FMH)
EIBCOMPL	Whether the RECEIVE command just issued used all the input data, or more RECEIVEs are required (see "Chaining output data" on page 539).
EIBCPOSN	Cursor position at time of last input operation (3270s only)
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 525).
EIBSIG	Whether the terminal has sent a SIGNAL
EIBTRMID	(CICS) identifier of the terminal

Chapter 28. Using SNA

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

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

z/OS Communications Server for SNA supports the chaining of outbound and 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 need 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.

© Copyright IBM Corp. 1989, 2013

Handling logical records

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 z/OS Communications Server for SNA, 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 can be the only SEND) to complete before terminating your task. Consequently, if an error does occur, it can 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 can 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 must 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).

The header 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 examines the PROFILE definition under which the transaction is executing to decide whether to remove the FMH 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 with 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 the z/OS Communications Server for SNA. 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 Communications Server 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.

Chapter 29. 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 705.)

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 671. Sequential terminals are 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

© Copyright IBM Corp. 1989, 2013

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 input can be a CESF GOODNIGHT transaction, which signs the terminal off and puts it out of service.

Alternatively, the last transaction executed can put the terminal out of service after its final output. You cannot normally enter further input from a sequential terminal once CICS has processed its original input, without putting it out of service.

Chapter 30. Using batch data interchange

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 *IBM 3767/3770/6670 Guide*. Figure 122 gives an overview of batch data interchange.

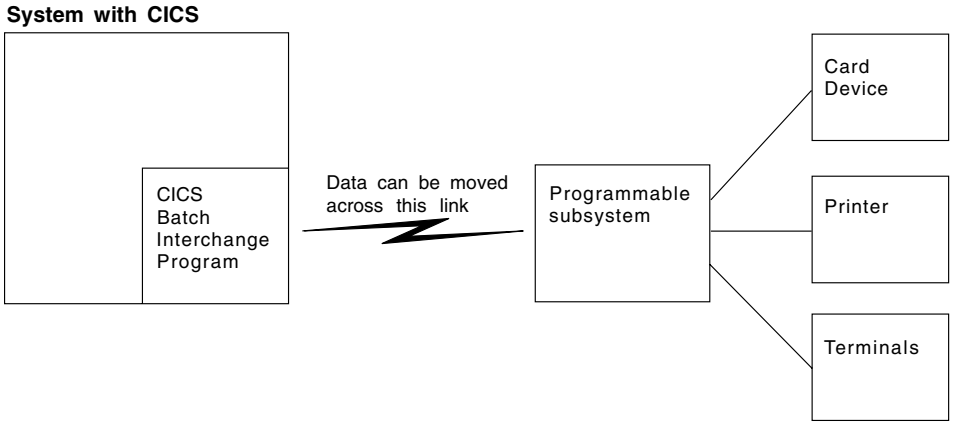


Figure 122. CICS batch data interchange

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.

© Copyright IBM Corp. 1989, 2013

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 "Dealing with exception conditions" on page 349 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.

When you select a destination 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.

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 (DEFRESP option)

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 (NOWAIT option)

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 31. Terminal control: design for performance

This list shows you what you need to consider for terminal control when designing for performance.

- **Minimize the 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 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.

- **Use only one physical SEND command per screen.**

  It is typically 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 can 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).

© Copyright IBM Corp. 1989, 2013

Chapter 32. The 3270 family of terminals

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 IBM 3270 Data Stream Programmer's Reference. IBM 3270 Data Stream Device Guide also contains important information. It is primarily intended for programmers using terminal control, but contains information that might be helpful for BMS programmers as well. BMS support for a few special features is discussed in "Support for special hardware" on page 770.

Although this discussion 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 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 42, "Printing and spool files," on page 659.

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 562.

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 553 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 (1 byte), a write control character or WCC (1 byte) and display data (variable number of bytes).

© Copyright IBM Corp. 1989, 2013

The WCC and display data are not always present; the write command determines whether a WCC follows and whether data can 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 running transaction. 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

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 send the data, 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). The **erase/write alternate** command selects the alternate size.

CICS uses the **write** command to send data unless you include the ERASE option on your **SEND** command:

- If you specify ERASE, CICS uses the PROFILE definition associated with the running transaction to determine whether to use **erase/write** or **erase/write alternate**.
- If you specify ERASE DEFAULT, CICS uses the **erase/write** command and sets the screen to default size.
- If you specify ERASE ALTERNATE, CICS uses the **erase/write alternate** and sets the screen to alternate size).

The **erase unprotected to address** command causes a scan of the buffer for unprotected fields; these fields are defined more precisely in "3270 field attributes" on page 554. 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 in "Merging the symbolic and physical maps" on page 715. No WCC or data follows this command; you send only the command.

The **write structured fields** command 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.

For more information, see IBM 3270 Data Stream Device Guide.

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 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 tags are explained in "3270 field attributes" on page 554).

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 565.

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" on page 554 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
(Typically 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 graphical 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 39 on page 556 lists the seven extended attributes and the values they can take.

3270 field attributes

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 713 for help from CICS in this area.)

Protection

There are four choices for the protection attribute, using up 2 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 the location of the next keystroke; for more information about this, see "Input from a 3270 terminal" on page 562.) 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 changes the field contents. The MDT bit is 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 on the MDT 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 can 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.

2 bits are used for display intensity, which allows one more value to be expressed than the three listed previously. For terminals that have either of the associated special hardware features, these same 2 bits are used to determine whether a field is light-pen detectable or cursor selectable. Because there are only 2 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 can be either. “Cursor and pen-detectable fields” on page 774 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 previously described, some 3270 terminals have extended attributes as well. The following table lists the types of extended attributes in the first column and the possible values for each type in the second column.

Table 39. 3270 extended attributes

Attribute type	Values
Extended color	Blue, red, pink, green, turquoise, yellow, neutral
Extended highlighting	Flashing, 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
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 before use. You can use a terminal control SEND command to do this.
SO/SI creation	Shift characters indicating double-byte characters can 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 Chapter 27, "Finding out about your terminal," on page 535.

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 747. "The set attribute order" on page 559 describes the format of such a sequence.

Orders in the data stream

Writing to a 3270 using terminal control commands you need to format the outbound data to express the attributes, position, and contents of a field.

About this task

If you are using BMS, all this is done for you, and you can move on to "Input from a 3270 terminal" on page 562.

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 1 byte long and typically 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 123.

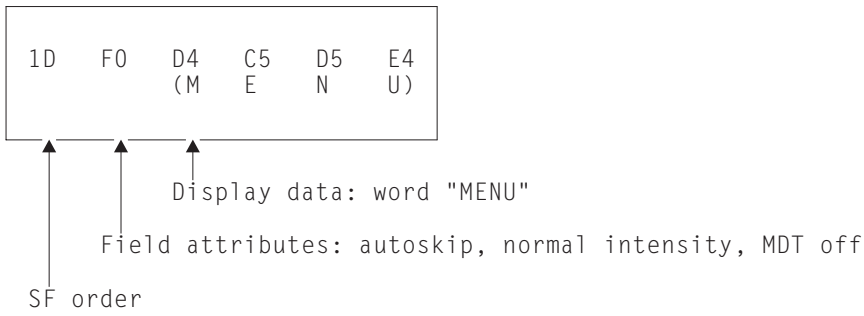


Figure 123. Field definition using SF order

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 124.

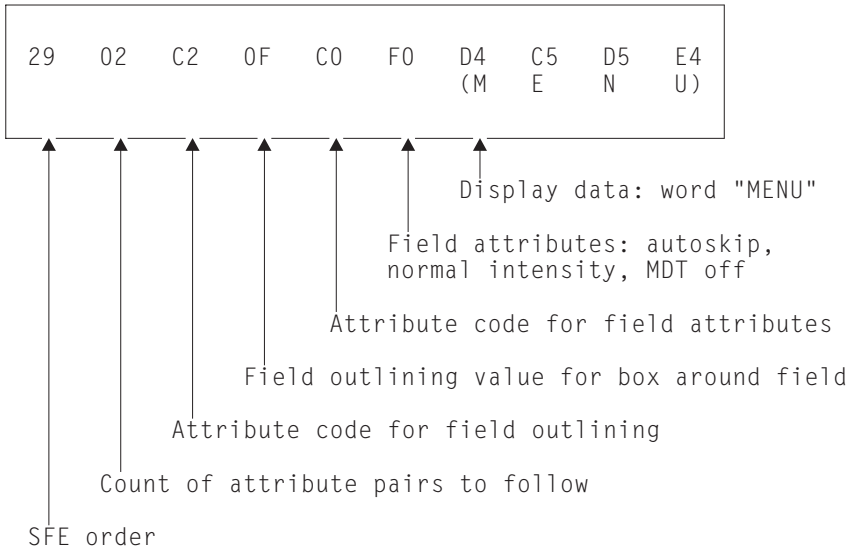


Figure 124. Field definition using SFE order

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 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 125.

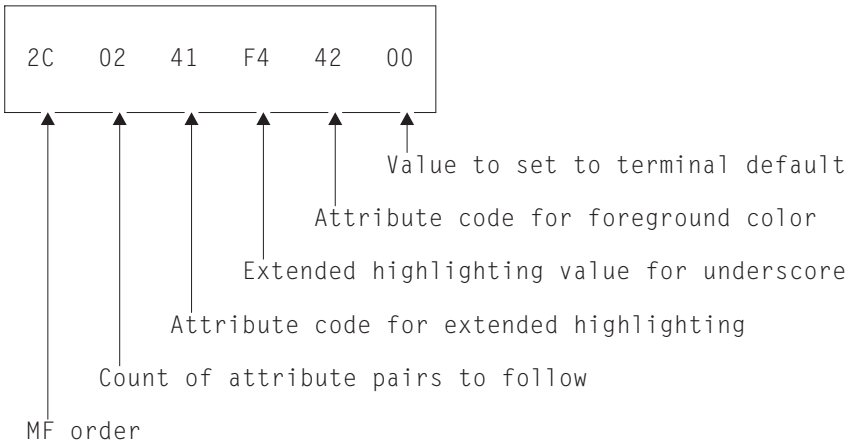


Figure 125. Changing field attributes within an MF order

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 126.

Figure 126. 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. 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 bit 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 6 bits of the rightmost byte of the address and B'000001' in the low-order 6 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 flash, you need the sequence in Figure 127.

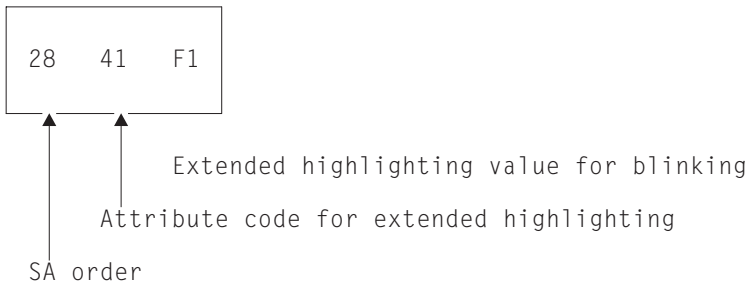


Figure 127. SA sequence to make a character flash

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 example

The outbound data stream example shows the data stream required to paint a particular 3270 screen, to reinforce the explanation of how the data stream is built.

Figure 128 shows an example screen that is part of an application that tracks 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 128. 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 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), 6 positions long
4. Label field, “Tag. No:”, at line 3, column 21
5. An input field (tag number) at line 3, column 31, 8 positions long
6. Label field, “State:”, at line 3, column 40
7. An input field (state), at line 3, column 49, 2 positions long
8. A field to mark the end of the previous (state) input field, at line 3, column 52

Table 40 shows the outbound data stream:

Table 40. 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 ended, so you do not have to precede it with an SBA sequence.
36	X'02'	Count of attribute *types* 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.

Table 40. 3270 output data stream (continued)

Bytes	Contents	Notes
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 enter data, 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 they try 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).
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 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; CICS supplies the write command and the WCC from options on your SEND command.

Input from a 3270 terminal

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, or attention key.

Data keys

The data keys include all the familiar letters, numbers, punctuation marks, and special characters. Pressing a data key 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, signals that the buffer is ready for transmission to the host. If the host has issued a read to the terminal, the typical situation in CICS, transmission occurs at this time.

There are five types of attention key:

- ENTER
- F (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 previously require special hardware, and in most cases the screen (buffer) must be set up appropriately beforehand. We talk about these features in “Support for special hardware” on page 770. For this section, 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 function 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 can 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 function 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 function 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 565.

Reading from a 3270 terminal

There are two basic read commands for the 3270, read buffer and read modified.

For either command, the inbound data stream starts with a 3 byte **read header** consisting of:

- Attention identifier (AID), 1 byte
- Cursor address, 2 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 "Support for special hardware" on page 770. Input from a **program attention** key on an SCS printer is also an exception; see "SCS input" on page 665 for a description of that data stream.

Inbound field format

If you are using terminal control commands, you must be aware of the format in which the 3270 transmits data.

If you are using BMS, you can skip to "Unformatted mode" on page 565, because BMS translates the input for you.

Each modified field comes in as follows:

- SBA order
- 2 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 must 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 773 for details.

Input data stream example

This example shows the input data stream created using the data entry screen from the previous outbound data stream example.

To illustrate an inbound data stream, we assume that an operator using the screen shown in Figure 128 on page 559 did the following:

- Put "123456" in the employee identifier field
- Put "ABC987" in the tag number
- Pressed ENTER, omittinglast the state field

Here is the resulting inbound data stream:

Table 41. 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 previously used 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 the remaining null positions were not completed by an operator.

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 3 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 exception conditions” on page 733. 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 its sets the buffer to nulls, erasing all the attributes bytes that demarcate fields.

Part 9. Interval control and task control

Controlling the timing and activities of tasks.

© Copyright IBM Corp. 1989, 2013

Chapter 33. Interval control

The CICS interval control services provide functions that are related to time.

Java and C++

The application programming interface described here 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 Java programming using JCICS in Developing applications and the JCICS Javadoc documentation. For information about C++ programs using the CICS C++ classes, see Using the CICS C++ OO foundation classes in Developing applications.

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 21st century.

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 CICS Interdependency Analyzer for more information about this utility and “Affinity” on page 312 for more information about transaction affinity.

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 with the REQID option. If you do not, CICS assigns (for POST and START commands only) a unique request

© Copyright IBM Corp. 1989, 2013

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.

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.

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 can 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 can therefore be earlier than the current time. TIME can 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 can 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.

Note:

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 the **INQUIRE SYSTEM**. During a normal shutdown, your task could run at the same time as the PLT programs with consequences known only to you.

Chapter 34. 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 here 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 programming using JCICS in Developing applications and the JCICS Javadoc documentation. For information about C++ programs using the CICS C++ classes, see Using the CICS C++ OO foundation classes in Developing applications.

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 574.
- 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

© Copyright IBM Corp. 1989, 2013

subsequently cleared, your task is not suspended and continues to run as though WAITCICS or WAIT EXTERNAL had not been issued.

WAIT EXTERNAL typically 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 CICS Interdependency Analyzer for more information about this utility and “Affinity” on page 312 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 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 want.

Note: Only one task can wait on a given ECB at any one time.

The example in Figure 129 on page 575 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 130 on page 575.

In theory, these tasks could exchange data through the temporary storage queue for ever. In practice, some code would be included to close down the process in an orderly way.

```
Task A                                          Task B
Delete temporary storage queue
Clear ECB1 (set to X'00000000')
Clear ECB2
EXEC CICS START task B ( pass addresses         EXEC CICS RETRIEVE
of PTR_ECB1_ADDR_LIST and                         (addresses passed)
PTR_ECB2_ADDR_LIST

LOOP:                                           LOOP:
 EXEC CICS WAITCICS                              Write to TS queue
  ECBLIST(PTR_ECB1_ADDR_LIST)                    Post ECB1 (set to X'40008000)
  NUMEVENTS(1)                                   EXEC CICS WAITCICS
 Clear ECB1                                       ECBLIST(PTR_ECB2_ADDR_LIST)
 Read TS queue                                    NUMEVENTS(1)
 Process data
 Delete TS queue
 Write to TS queue
 Post ECB2                                       ClearECB2
Go to start of loop                              Read TS queue
                                                 Process data
                                                 Delete TS queue
                                                 Go to start of loop
```

Figure 129. Two tasks using WAITCICS to control access to a shared resource

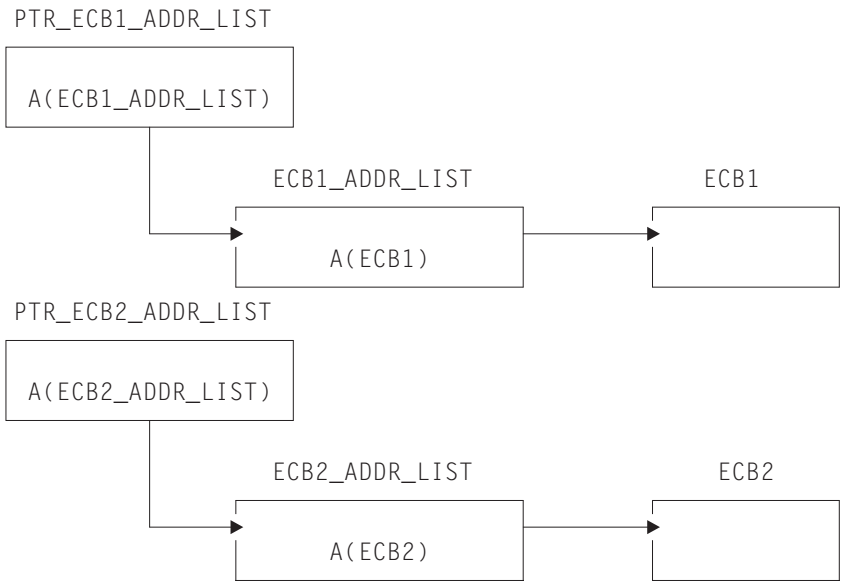


Figure 130. ECB pointers used by WAITCICS example

"Dealing with exception conditions" on page 349 describes how the exception conditions that can occur during processing of a task control command are handled.

Part 10. Storage protection and transaction isolation

Protecting CICS code, control blocks, and other transactions from being overwritten by an application program.

© Copyright IBM Corp. 1989, 2013

Chapter 35. CICS storage protection and transaction isolation

Storage control is affected by storage protection and transaction isolation. Storage protection protects CICS code and control blocks from applications, and transaction isolation protects tasks from each other.

The subsystem storage protection facility enables you to prevent your application programs from accidentally overwriting CICS code and control blocks. It does not provide protection against deliberately overwriting CICS code or control blocks. CICS cannot prevent an application from obtaining the necessary access (execution key) to modify CICS storage.

Transaction isolation extends this storage protection to provide protection for transaction data. If an application program accidentally overwrites the transaction data of another transaction, it 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 use CICS system initialization parameters to specify whether to use storage protection facilities. For information about the relevant parameters, see Storage protection in Improving performance.

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 here is the 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 programming using JCICS in Developing applications and the JCICS Javadoc documentation. For information about C++ programs using the CICS C++ classes, see Using the CICS C++ OO foundation classes in Developing applications.

CICS storage control facility

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. “Design for performance” on page 252 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

© Copyright IBM Corp. 1989, 2013

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 CICS Interdependency Analyzer. See CICS Interdependency Analyzer for information about this utility and see "Affinity" on page 312 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 section describes:

- Chapter 35, "CICS storage protection and transaction isolation," on page 579
- "Defining the storage key for applications" on page 583
- "Selecting the execution and storage key" on page 586
- "Protection with transaction isolation" on page 589
- "MVS subspaces" on page 591

Storage protection

CICS allows you to run your application programs in either user-key or CICS-key storage. 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 131.

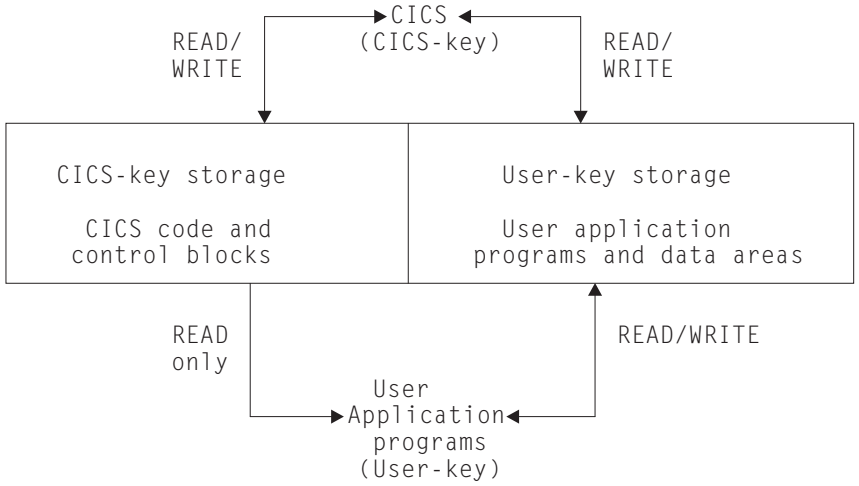


Figure 131. Protecting CICS code and control blocks from user application programs

Related information:

"Storage categories"
When you are running with the storage protection facility active, there are two types of storage in which you can run application programs: CICS-key storage and user-key storage.

STGPROT system initialization parameter in Reference -> System definition

Storage categories

When you are running with the storage protection facility active, there are two types of storage in which you can run application programs: CICS-key storage and user-key storage.

CICS-key storage
Used for most 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
Used for most application programs and their data areas.

There are two associated execution modes:

1. Many CICS system programs run in **CICS key**. A program running in CICS-key has read-write access to both CICS-key and user-key storage.
2. Application programs normally execute in **user key**. A program running in user-key has read-write access to user-key storage, but only read access to CICS-key storage.

The terms "user key" and "CICS key" therefore apply both to categories of storage and to the execution of programs. These terms are reflected in the resource definition keywords used in TRANSACTION definitions. See Define FILE attributes in Reference -> System definition 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. 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 designed to provide special function in support of user applications. Such programs are 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 587.

The storage protection facility does not protect CICS-key code and control blocks from being overwritten by programs that you choose to run in CICS key.

Defining the execution key

To run your programs in CICS key, you use the execution key parameter (EXECKEY) on the program resource definition. See "Selecting the execution and storage key" on page 586 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

You can 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.

About this task

Depending on the data area, you select the storage key by using one of the following:

- System initialization parameters
- Resource definition option
- Options on the GETMAIN commands

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 these areas 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.

Related reference

CWAKEY system initialization parameter in Reference -> System definition

TCTUAKEY system initialization parameter in Reference -> System definition

Task-lifetime storage

You can 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.

To specify whether user-key or CICS-key storage is used, you use the TASKDATAKEY option on the transaction resource definition. This option 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 the following storage requests:
  - Explicit requests as a result of GETMAIN commands
  - Implicit requests as a result of a CICS command that uses the SET option

For information about how to specify the TASKDATAKEY parameter, see TRANSACTION attributes in Reference -> System definition.

For an illustration of what TASKDATAKEY controls for both task lifetime storage and program working storage, see Figure 132 on page 585.

For programming information about EXEC CICS commands, see CICS command summary in Reference -> Application development .

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 commands

The GETMAIN and GETMAIN64 commands provide 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, you can use these options for application programs that are executing with TASKDATAKEY(CICS) specified to obtain user-key storage for passing to, or returning to, a program that is executing in user key.

In a program that is defined with EXECKEY(CICS) that issues GETMAIN commands to obtain CICS-key storage, such storage can be freed explicitly only if the FREEMAIN or FREEMAIN64 command is issued by a program that is also defined with EXECKEY(CICS). If an application program that is defined with EXECKEY(USER) attempts to free CICS-key storage by using FREEMAIN commands, CICS returns the INVREQ condition. However, an application can free user-key storage with FREEMAIN commands regardless of the EXECKEY option.

CICS frees all task-lifetime storage that is acquired by an application at task termination, whether it in CICS key or user key. You can also specify STORAGECLEAR(YES) on this option of the associated transaction resource definition. This clears the storage so that another task cannot view sensitive data accidentally.

For programming information about commands, see CICS API commands in Reference -> Application development. For information about defining resources,

see An overview of resource definition in Product overview.

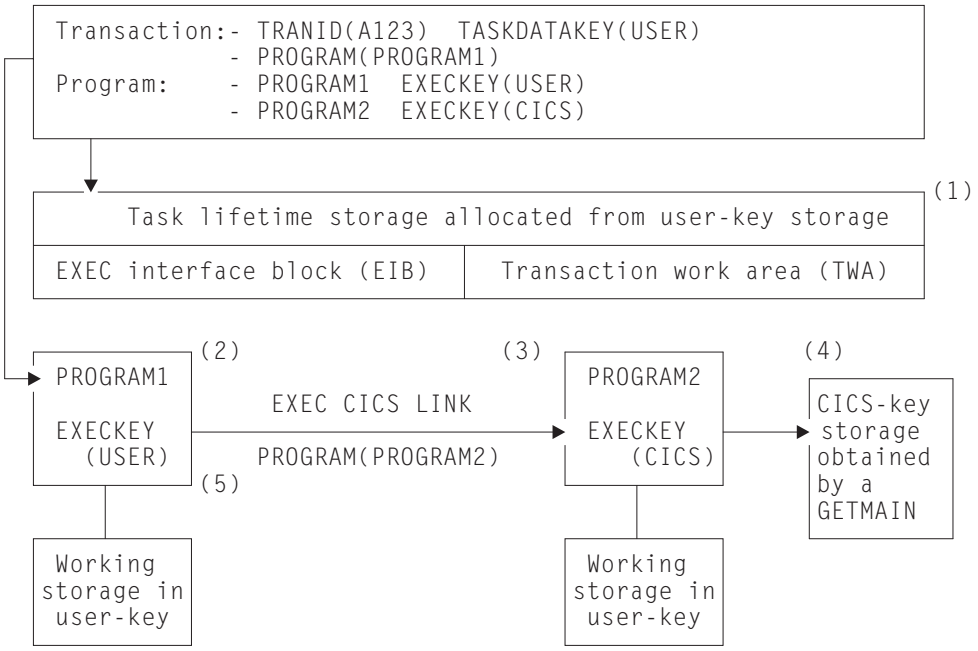


Figure 132. Illustration of the use of the TASKDATAKEY and EXECKEY options

In this example, transaction A123 is defined with TASKDATAKEY(USER) and PROGRAM(PROGRAM1). PROGRAM1 is defined with EXECKEY(USER) and PROGRAM2 is defined with EXECKEY(CICS). PROGRAM1 links to PROGRAM2, which uses a GETMAIN request to obtain CICS-key storage.

Notes:

1. The TASKDATAKEY option ensures that the transaction work area (TWA) and EXEC interface block (EIB) are allocated from user-key storage, as 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 using 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 CICSDATAKEY 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.

About this task

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 42 summarizes the various combinations of options.

Table 42. 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

Some application programs need to be defined with EXECKEY(CICS) because they need to use certain facilities that are listed later.

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 most 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.

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 here:

- 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 can 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 3, "Developing COBOL applications," on page 23, Chapter 4, "Programming in C and C++," on page 45, and Chapter 5, "Developing PL/I applications," on page 55 for information about which of these should not be used in CICS application programs. It is possible that some of these functions might have worked in previous releases of CICS, or at least might 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 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 261 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 264 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 like CICS system code. For example, you might 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
  - Custom EP adapters handling synchronous events

  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 591), depending on the current mode when CICS gives control to the exit program. They can also modify any application storage. The URMs and PLT programs execute in base space.

For programming information about the execution of GLUEs, TRUEs, URMs, and PLT programs in a CICS region running with storage protection, see Exit programs and the CICS storage protection facility in 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 CICS Interdependency Analyzer 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 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 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 Dealing with transaction abend codes in the *CICS Problem Determination Guide* for more information about the storage protection exception conditions.

Protection with transaction isolation

In addition to being able to specify the storage and execution key for user transactions, you can also specify whether you want transaction isolation. Transaction isolation is built on top of storage protection, which means that STGPROT=YES must be specified. Transaction isolation uses parameters introduced by storage protection, these being EXECKEY, and TASKDATAKEY.

You can control transaction isolation globally for the whole CICS region with 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.

A user-key program invoked by transaction A (TXNA) can 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.

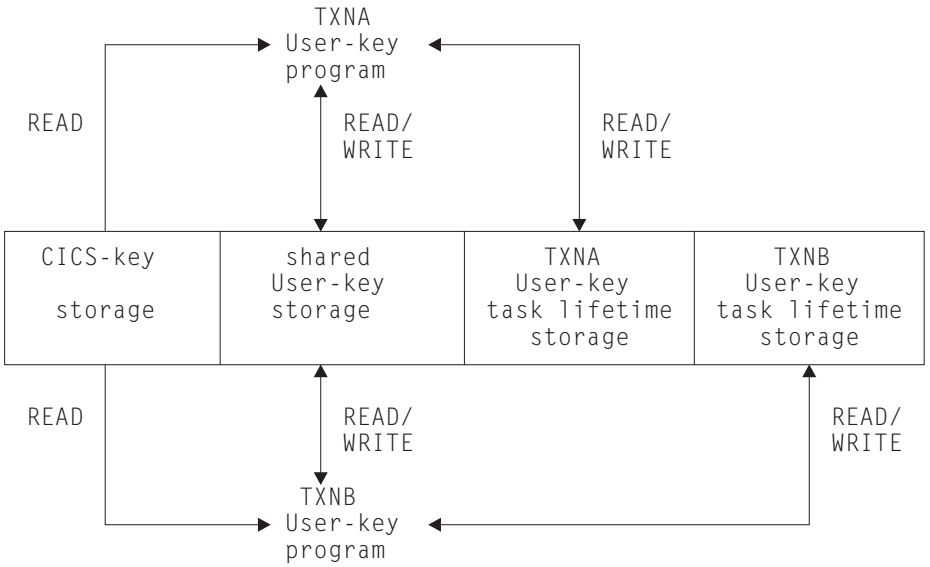


Figure 133. Two transactions defined as ISOLATE(YES)

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).

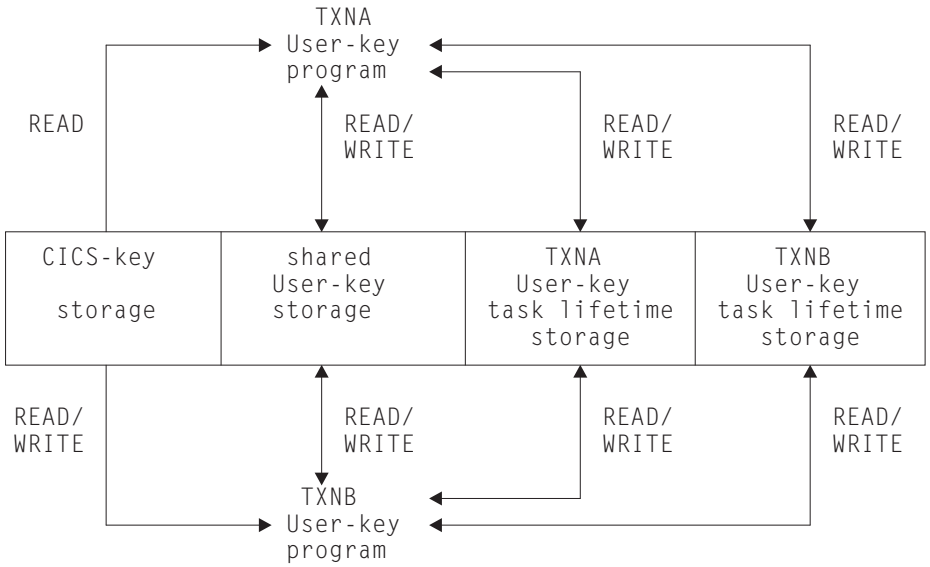


Figure 134. Two transactions defined as ISOLATE(NO) with read/write access to each other's task lifetime storage

User-key storage is not protected from CICS-key programs, even when you set ISOLATE(YES).

MVS subspaces

You can use the subspace group facility in MVS 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.

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. 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. Therefore, a user transaction is limited to its own “view” of the address space.

Programs defined with EXECKEY(CICS) run in the base space.

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 might need special definition if they do any of the following:

- Issue MVS macros directly
- Modify CICS control blocks
- Have a legitimate need for one task to access, or share, another task's storage

Some existing transactions can share task-lifetime storage in various ways, and this sharing might 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.

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 must 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 must 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 previous releases of CICS should work under CICS Transaction Server for z/OS, Version 5 Release 1 if defined with ISOLATE(NO) to ensure that they run in the common subspace. This provides support for upgrading, 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.

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 CICS Interdependency Analyzer for more information about this utility.

Part 11. Transient data and temporary storage

Controlling transient data and temporary storage queues.

© Copyright IBM Corp. 1989, 2013

Chapter 36. Transient data control

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 *intrapartition* or *extrapartition* transient data queues.

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 37, "Temporary storage control," on page 601 for more information about temporary storage.)

Java and C++

The application programming interface described here is the CICS API, which is not used in Java programs. For information about Java programs using the JCICS classes to access transient data services, see Java programming using JCICS in Developing applications and the JCICS Javadoc documentation. For information about C++ programs using the CICS C++ classes, see Using the CICS C++ OO foundation classes in Developing applications.

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

© Copyright IBM Corp. 1989, 2013

- 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 must not include the FREE=CLOSE operand. If you do specify FREE=CLOSE, attempts to read the queue after the queue has been closed and then reopened 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 TDQUEUE resource definitions 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.

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 execution of the earlier task has ended. The exact point at which a new ATI cycle begins depends on whether 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 can 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 sync point.

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 EIBTRMID. 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 might create inter-transaction affinities that adversely affect your ability to perform dynamic transaction routing. See "Affinity" on page 312 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.

Transactions that are initiated because of a transient data trigger-level transaction do not create previous hop data and are treated as executing at a point of origin. For more information about previous hop data, see Previous hop data characteristics.

Chapter 37. 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.

For an overview of the different temporary storage locations and the features available for temporary storage queues in each location, see Main temporary storage: Monitoring and tuning in Improving performance.

The CICS temporary storage control facility includes commands for the following purposes. On each of these commands, the TS keyword can be omitted; temporary storage is assumed if it is not specified.

- 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).

For more information about how programmers can use temporary storage queues, see "Temporary storage queues" on page 245.

Exception conditions that occur during execution of a temporary storage control command are handled as described in "Dealing with exception conditions" on page 349.

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 CICS Interdependency Analyzer for more information about this utility and "Affinity" on page 312 for more information about transaction affinity.

Java and C++

The application programming interface described here is the 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 programming using JCICS in Developing applications and the JCICS Javadoc documentation. For information about C++ programs using the CICS C++ classes, see Using the CICS C++ OO foundation classes in Developing applications.

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

© Copyright IBM Corp. 1989, 2013

should be accessed using the READQ TS command with the ITEM option; not doing so can 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 wanted. 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 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 12. 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).

© Copyright IBM Corp. 1989, 2013

Chapter 38. CICS documents and document templates

Application programs can create documents and place data into them using commands in the EXEC CICS DOCUMENT application programming interface. Document templates are portions of a document which can be created offline, or in another CICS program, and used by the application program to create the document.

Documents and document templates are most commonly used to produce Web pages provided by CICS Web support. They can contain HTML that is used as the body of an HTTP request or response. However, they are not limited to this use.

Documents

You can create an empty document in an application program using the EXEC CICS DOCUMENT CREATE command, and then build the contents with subsequent DOCUMENT INSERT commands. Or you can use DOCUMENT CREATE to create and build a document in one step. You can create a document using data specified by your application program, or using a document template, or using another document. The document handler returns a token (DOCTOKEN), which is used to identify the document on subsequent calls.

When a document has been created, the contents can be extended by issuing one or more DOCUMENT INSERT commands. Again, you can add data specified by your application program, or a document template, or another document. You can also insert bookmarks into the document between blocks of data, and use the bookmarks to add or replace data in the middle of the 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 command. The application can then save this copy to a location of its choice, such as a temporary storage queue. This copy can then be used to re-create the document.

Document templates

Document templates can be retrieved from several different sources, to suit the way they are used in the application program. The source of a document template can be any one of the following:

- A partitioned data set
- A CICS program
- A CICS file
- A z/OS UNIX System Services file
- A temporary storage queue
- A transient data queue
- An exit program

Document templates are defined using DOCTEMPLATE resource definitions, which specify the source of the document template.

© Copyright IBM Corp. 1989, 2013

Document templates can contain static data, and also symbols. Symbols represent variable data that is resolved at the time the template is added to the document, that is, at the time the DOCUMENT CREATE or DOCUMENT INSERT command is issued. The values to be substituted for the symbols are specified by the application program using the SYMBOLLIST option of the DOCUMENT CREATE command, or the SYMBOLLIST or SYMBOL options of the DOCUMENT SET command.

Document templates can also contain embedded commands to set default values for symbols, identify symbols, and embed another template.

As well as being used by application programs, document templates can be specified in a URIMAP definition to provide a static response to a Web client's HTTP request, with no application program needed. CICS produces a response using the document template as the body of the Web page.

Symbols and symbol lists

Symbols are used in document templates to enable application programs to customize documents with data appropriate to the current task, such as a user name or order number. 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 command is issued.

Each symbol has a name, and it can have a value assigned to it. When a document template containing symbols is inserted into a document, the CICS document handler uses the values assigned to the symbols for the process of **symbol substitution**, where the symbol in the document template is replaced by its value. For example, if the symbol ORDER_NUMBER is assigned a value of `0012345`, and used in a document template as follows:

```
Thank you! Your order number is &ORDER_NUMBER;.
```

the finished document after symbol substitution will be:

```
Thank you! Your order number is 0012345.
```

A **symbol reference** is included in a document template at the place where the value of the symbol is to be substituted in the text of the completed document. The symbol reference consists of the symbol name, preceded with an ampersand (&) and terminated with a semicolon (;). You can use a symbol in any location within the document template. You can also include a default value for the symbol at any location in the document template, using the embedded template command #set.

An application program that creates a document using the document template needs to assign values to the symbols that are to be substituted when the template is used. You can do this using the DOCUMENT SET command or the DOCUMENT CREATE command.

You can specify values for individual symbols using the SYMBOL and VALUE options on the DOCUMENT SET command. Alternatively, you can define several symbols in a single command by specifying a **symbol list**. A symbol list is a character string consisting of one or more definitions, each containing a name, an equals sign, and a value, with single byte separators (by default, an ampersand). It can be specified using the SYMBOLLIST option on the DOCUMENT CREATE or DOCUMENT SET command. For example, here is a symbol list that specifies the value '0012345' for the symbol ORDER_NUMBER, and the value 'Germany' for the symbol COUNTRY:

```
ORDER_NUMBER=0012345&COUNTRY=Germany
```

The values that have been assigned to symbols are held in a **symbol table**. There is a symbol table associated with each document template. When the #set command is used, or when an application provides a value for a symbol, the symbol name and its associated value are added to the symbol table. If the symbol already exists in the symbol table, its value is replaced by the new value. A symbol definition provided by an application overrides a default value provided by a #set command.

When a document template is inserted into a document by the DOCUMENT CREATE or DOCUMENT INSERT command, symbol substitution is carried out for all the symbols that exist in the symbol table, using their current values as specified in the symbol table. If the symbol has not been specified by the application program or by a #set command, no symbol substitution takes place, and the finished document includes the symbol name or the #echo command specifying the symbol.

When you use a template containing symbols, you need to specify the values you require for the symbols **before** inserting the template into a document. If you insert a template, but you have not assigned values to the symbols in it, the symbols will never be substituted. (This can occur if you create a document from a template without specifying a symbol list or symbol values.) After you have inserted the template into a document, you cannot change the values that CICS has put in the document in place of the symbols. If you specify values for symbols after inserting the template, your values are placed into the symbol table and used the next time that the template is inserted into a document, but your changes do not affect the values that have already been inserted into the document.

Data format for symbols and symbol lists

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. (Web pages are the most common use for CICS document templates.)

The format of url-encoded data is a sequence of name and value pairs, delimited by ampersands, for example:

```
firstname=Irving&lastname=Berlin
```

The values are normally data which the user has entered in the form.

However, if the data entered in the form contains characters that have a special meaning in the format of url-encoded data, the characters entered in the form need to be distinguished from those used to delimit the name and value pairs. To achieve this, a process of character escaping is performed. An escaped character is a character that has been replaced by the three-character sequence %xx, where xx is the hexadecimal value of the ASCII encoding of the character.

The ampersand (&), the equals sign (=), and also the percent sign (%), because it is used to introduce the escape sequence, must all be escaped in url-encoded data. Because the space character is often used as a delimiter, it is always escaped too. However, because spaces are so common, the url-encoded data type allows spaces to be encoded as plus signs (+), rather than as an escape sequence. This in turn means that any plus signs entered in the form must also be escaped.

For example, if a form contains the following names and values:

```
sum         8+11=19
rate        19%
composers   George & Ira Gershwin
```

the escaped url-encoded representation of this data is:

```
sum=8%2b11%3d19&rate=19%25&composers=George+%26+Ira+Gershwin
```

The space characters in the values have been encoded as plus signs, and the plus sign, equals sign, percent sign, and ampersand in the values have been encoded as escaped sequences.

A symbol list which you specify using the SYMBOLLIST option of the EXEC CICS DOCUMENT CREATE or DOCUMENT SET commands contains, in principle, a list of name and value pairs in the url-encoded format. However, CICS has extended this syntax in the following ways:

- Spaces do not have to be escaped. A space can be represented as a single space, or as a plus sign, or as the escape sequence %20.
- The delimiter between the name and value pairs does not have to be an ampersand. You can specify an alternative delimiter using the DELIMITER option of the command.

Normally, unescape processing is carried out for the values of symbols when they are put into the symbol table, so any characters specified using special coding are converted into the intended character. For example, plus signs are converted to spaces, and escape sequences are converted to the correct characters. However, if you specify the UNESCAPED option on the DOCUMENT CREATE or DOCUMENT SET command, no conversion takes place, and the symbol values are put into the symbol table exactly as you entered them.

Symbols in HTML comments

Within HTML comments in document templates, symbols are normally ignored, and the CICS document handler does not carry out symbol substitution. HTML comments are delimited by the opening markup `<!--` and the closing markup `-->`.

If you want to have symbol substitution occur within an HTML comment in a document template, you can use symbols in place of the opening and closing delimiters of the comment, and use the #set command in the document template to provide the opening and closing delimiters as the values for those symbols. If you do this, CICS carries out symbol substitution for the whole of any comment around which you use the symbols in place of the delimiters.

For example, a document template could include:

```
<!--#set var=OC value='<!--'-->
<!--#set var=CC value='-->'-->

&OC; A comment containing my text &SYM; &CC;
```

If the application program assigns a value of 'Example text' to the symbol SYM, CICS substitutes the value of that symbol and also of the symbols OC and CC, producing the HTML output:

```
<!-- A comment containing my text Example text -->
```

Caching and refreshing of document templates

To improve performance, the CICS document handler caches a copy of most document templates. When applications reference the template, they use the cached copy, improving performance. You can refresh the cached copy at any time if the document template changes. You can also phase in a new copy of programs and exit programs that are defined as document templates.

CICS always caches a copy of the following types of document template:

- Templates in a partitioned data set
- Templates in a CICS file
- Templates in a z/OS UNIX System Services file
- Templates in a temporary storage queue
- Templates in a transient data queue

When one of these types of document template is installed individually while CICS is running, it is read into the CICS document handler's storage. Requests from applications to access the document template receive the cached copy of the template, so CICS does not need to access the location where the document template is stored each time. Document templates that are installed during CICS startup are not cached at that time; each of these document templates is cached when it is referenced for the first time by an application.

If you make changes to a document template that has been cached, you can refresh the cached copy of the document template using the CEMT or EXEC CICS SET DOCTEMPLATE NEWCOPY command. (Note that with the SET DOCTEMPLATE command, which is not part of the EXEC CICS DOCUMENT API, you need to specify the name of the DOCTEMPLATE resource definition which defines the document template, rather than the 48-character name of the template.)

For the types of document template listed above, the SET DOCTEMPLATE NEWCOPY command deletes the copy of the document template which is currently cached by the CICS document handler, and replaces it with a copy read from the location where the document template is stored. (For templates in a partitioned data set, CICS first performs a BLDL (build list) to obtain the most current directory information, and then rereads the member.) When a new cached copy has been created, subsequent requests to use the document template use the new copy. The new copy will be used by later requests within the same task, as well as requests in other tasks.

If the CICS system becomes short on storage, the document handler deletes some of the cached copies of document templates to attempt to relieve the storage constraint. The document templates to be deleted are selected in order of size, largest first, taking into account the time since the cached copy was created (so that newly created copies are not released immediately).

If the CICS system is restarted with a warm start, the document templates that were previously cached are not reloaded. The cache is repopulated as each document template is referenced for the first time by an application.

The CICS statistics collected for document templates show the number of times each document template is referenced, and the number of times a cached copy was made, refreshed, used and deleted.

Templates in CICS programs

Document templates retrieved from CICS programs are never cached by the document handler, because programs are already cached elsewhere in CICS.

For this type of document template, you can use the SET DOCTEMPLATE NEWCOPY command to phase in a new copy of the program. The command is equivalent to SET PROGRAM PHASEIN for the specified program. Subsequent requests to use the document template use the new copy, including later requests within the same task.

Templates in exit programs

For document templates generated by an exit program, the exit program specifies (in its exit parameter list) whether or not a copy of the document template should be cached by the document handler. The default is that the document template is not cached. Templates that change dynamically should not be cached, but if the template does not change, caching is suitable as it improves the performance of requests. If the exit program does specify caching, the cached copy is made when the document template is referenced for the first time by an application.

For this type of document template, you can use the SET DOCTEMPLATE NEWCOPY command to phase in a new copy of the exit program. The command is equivalent to SET PROGRAM PHASEIN for the specified exit program. When you issue the command, CICS deletes any cached copy of the document template, phases in the new copy of the program, and creates a new cached copy of the document template if the exit program specifies caching. The refreshed exit program can specify a different setting for whether or not caching should take place, and CICS honors the change.

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 CHARACTERSET option to make the document handler 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.

Chapter 39. Setting up document templates

Document templates are CICS resources, which you define using DOCTEMPLATE resource definitions.

The name of the template, which is used in EXEC CICS DOCUMENT API commands to refer to the template, is specified in the TEMPLATENAME attribute of the resource definition. The DOCTEMPLATE resource definition also specifies the source of the template, the format of the data (binary or EBCDIC), and whether CICS appends a carriage-return line-feed to each record in the template.

Templates in a partitioned data set

HTML templates that have been created from BMS map set definitions are stored in partitioned data sets.

Document templates stored in partitioned data sets have the attribute MEMBERNAME in their DOCTEMPLATE resource definitions.

CICS loads a copy of the template from the partitioned data set when you install the corresponding DOCTEMPLATE definition. If you modify the template in the partitioned data set while CICS is running, issue the SET DOCTEMPLATE NEWCOPY command in order to activate the changes. CICS first performs a BLDL (build list) to obtain the most current directory information, and then rereads the member.

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 stored in a partitioned data set are cached by the CICS document handler.

Templates in z/OS UNIX System Services files

z/OS UNIX System Services files can be defined as document templates. You might find that z/OS UNIX files are the most convenient type of document template to create and edit.

© Copyright IBM Corp. 1989, 2013

Document templates stored in z/OS UNIX files have the attribute HFSFILE in their DOCTEMPLATE resource definitions.

For CICS Web support, you can deliver az/OS UNIX file directly as a static response using a URIMAP definition, in which case the z/OS UNIX file does not need to be defined as a document template (although it may be). However, if you want to carry out symbol substitution, or a Web-aware application program needs to access the file, it must be defined as a document template.

In the DOCTEMPLATE definition, you need to provide the fully-qualified name of the file, which can be up to 255 characters in length. The CICS region must have permission to access z/OS UNIX, and it must have permission to access the directory containing the file, and the file itself. Giving CICS regions permission to access z/OS UNIX directories and files in *Java Applications in CICS* explains how to achieve this.

All the data in the file is used as the document template.

Templates stored in a z/OS UNIX file are cached by CICS. You can use the SET DOCTEMPLATE NEWCOPY command to delete the copy of the document template which is currently cached, and replace it with a copy read from the location where the document template is stored.

Templates in CICS 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 document template.

Which resource you use depends on:

- 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.

Document templates stored in these resources have the attributes FILE, TSQUEUE, and TDQUEUE, in their DOCTEMPLATE resource definitions.

If you are using a DOCTEMPLATE defined with the attribute TSQUEUE or TDQUEUE, two bytes (a carriage return and line feed) are added to the end of the DOCTEMPLATE buffer.

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

Extrapartition transient data queues are suitable for use as document templates. All records in the queue are read.

Intrapartition transient data queues are not suitable for use as document templates. 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.

CICS files that are to be used as document templates must be specified with the BROWSE attribute.

Templates stored in any of these formats are cached by CICS. You can use the SET DOCTEMPLATE NEWCOPY command to delete the copy of the document template which is currently cached, and replace it with a copy read from the location where the document template is stored.

Templates in CICS programs

A document template in a program has the least overhead for retrieving the template, although for other formats, caching also minimizes the overhead after the first request. Programs are an appropriate format to choose if the same template is used by several applications. A CICS program is most suitable where the template is not changed frequently, because a program cannot be easily edited and must be recompiled.

About this task

Document templates contained in CICS programs have the attribute PROGRAM in their DOCTEMPLATE resource definitions.

Templates contained in CICS programs are never cached by the CICS document handler. Programs are already cached by the CICS loader.

For this type of document template, you can use the SET DOCTEMPLATE NEWCOPY command to phase in a new copy of the program. The command is equivalent to SET PROGRAM PHASEIN for the specified program. Subsequent requests to use the document template use the new copy, including later requests within the same task.

To code a program which contains a template:

Procedure

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 want to include in your template.
   c. An END statement.
   d. Invocations for the DFHDHTL macro, to generate a prolog and epilog for the document template. “DFHDHTL - program template prolog and epilog macro” on page 616 documents this macro.

   For example:

```
DOCTPROG CSECT
DOCTPROG AMODE 31
DOCTPROG RMODE ANY
         DFHDHTL TYPE=INITIAL,ENTRY=WKLYHDR
         DC    CL4'<HR>'
         DC    CL29'<H2>Weekly Status Report</H2>'
         DFHDHTL TYPE=FINAL
         END   WKLYHDR
```

The DFHDHTL macro is used because the template program must be an exact multiple of eight bytes. If this is not the case, the binder might insert spurious binary characters at the end of the template, which can produce unpredictable output when the template is used in a document. The DFHDHTL macro creates the necessary padding.

2. Assemble and link edit the program into your CICS application program library. The name you give to the program can be different from the name of the entry point.
3. Create and install a DOCTEMPLATE resource definition which specifies the name of the program in the PROGRAM attribute.

What to do next

CICS will autoinstall the program on first reference, or you can create and install a PROGRAM resource definition.

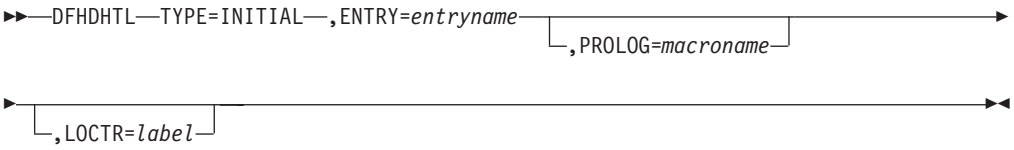
DFHDHTL - program template prolog and epilog macro

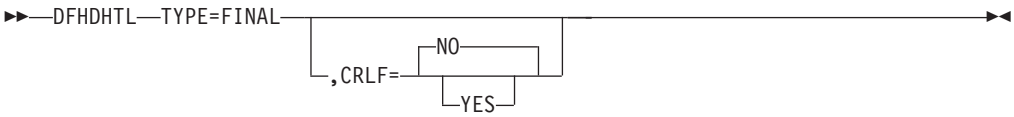
The DFHDHTL macro is used to generate the prolog and epilog for document templates contained in a CICS program (with the attribute PROGRAM in the DOCTEMPLATE resource definition).

These templates are defined as occupying the area of a load module between its entry point and the end of the module. However, if the source of the template program does not generate an exact multiple of eight bytes, the linkage editor or binder might insert spurious binary characters, which have an unpredictable effect when transmitted to a Web browser. The DFHDHTL macro helps to ensure that the template load module is always an exact multiple of eight bytes.

The prolog created by TYPE=INITIAL generates a module eyecatcher by invoking a user-specified macro (defaulting to DFHVM), and a new location counter (LOCTR) which will track the body of the template.

The epilog created by TYPE=FINAL calculates the length of the template so far, and if it is odd, generates a single blank character. This ensures that the entry point is on a halfword boundary. The epilog then rounds up the module size by reverting to the original LOCTR of the prolog, and generating sufficient characters before the entry point to make the module length a multiple of eight bytes.





TYPE={INITIAL | FINAL}
Specifies whether a prolog (INITIAL) or an epilog (FINAL) is to be generated.

ENTRY=*entryname*
Specifies the label for the entry of the module. This is the point at which the template starts.

PROLOG=*macroname*
Specifies a macro to be invoked to generate module identification data in the prolog. The default is DFHVM.

LOCTR=*label*
Labels the internally generated location counter (LOCTR) statement which separates the generated template from the prolog code. The default is an internally generated name.

CRLF=NO | YES
Specifies whether the epilog code is to terminate with a closing CRLF (carriage return line feed). The default is NO.

Templates in exit programs

An exit program is most suitable where the contents of the template change dynamically, or where 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.

Templates contained in an exit program are cached by CICS only if you specify this in the exit parameter list for the program. The default is that the templates are not cached. If the exit program does specify caching, the cached copy is made when the document template is referenced for the first time by an application.

You can use the SET DOCTEMPLATE NEWCOPY command to phase in a new copy of the exit program. The command is equivalent to SET PROGRAM PHASEIN for the specified exit program. When you issue the command, CICS deletes any cached copy of the document template, phases in the new copy of the program, and creates a new cached copy of the document template if the exit program specifies caching. The refreshed exit program can specify a different setting for whether or not caching should take place, and CICS honors the change.

Communication area for templates in exit programs

The communication area for an exit program which supplies a document template is mapped by a CICS-supplied copybook.

The supplied copybooks are:

- 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.

dhtx_cache_response
Use the characters '1' (cache) or '0' (do not cache) to specify whether or not the output from the exit program should be cached by the CICS document handler. The value of `dhtx_cache_response` is initialized to '0', so the default action is not to cache the response of the exit program, unless the exit changes this value.

When a document template is cached, subsequent requests receive the cached copy. The exit program is not called again as long as the cached copy is available, until the EXEC CICS SET DOCTEMPLATE NEWCOPY command is issued to refresh the exit program and the cached copy. A refreshed exit program can specify a different value for `dhtx_cache_response`, and CICS honors the change.

Templates that change dynamically should not be cached, but if the template does not change, caching is suitable as it improves the performance of requests.

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.

Using symbols in document templates

Use a symbol reference or #echo command in a document template at the place where the value of the symbol is to be substituted in the text of the completed document. You can use the #set command in the template to specify a default value for a symbol.

Before you begin

The name of a 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.

Procedure

- You can include a symbol reference in a document template by specifying the symbol name, preceded with an ampersand (&) and terminated with a semicolon (;). For example, the symbol ORDER_NUMBER could be specified in a template as follows:

  ```
  Thank you! Your order number is &ORDER_NUMBER;.
  ```

- You can also use the #echo command in the document template to specify a symbol. Specify the symbol name in the command, but do not use an ampersand and semicolon. For example, the symbol USER_NAME could be specified as follows:

  ```
  Welcome to the site,
  <!--#echo var=USER_NAME-->!
  ```

- You can use the #set command in the template to specify a default value for a symbol. For example, you could include the symbol USER_NAME and set a default value for it as follows:

  ```
  <!--#set var=USER_NAME value='New User'-->
  Welcome to the site,
  <!--#echo var=USER_NAME-->!
  ```

  or as follows:

  ```
  <!--#set var=USER_NAME value='New User'-->
  Welcome to the site, &USER_NAME;!
  ```

  When either of these templates is used with the default value for USER_NAME, the document appears as

  ```
  Welcome to the site, New User!
  ```

What to do next

An application program that creates a document using the document template needs to define values for the symbols that are to be substituted when the template is used. If you provide a default value for a symbol, this is used if the application does not define a value for that symbol. Remember that you must define values for the symbols before, or at the time when, you insert the template into the document. You cannot change the substituted values of the symbols after the template has been inserted.

Embedded template commands

The CICS document handler recognizes three commands that can be embedded in a document template. The three commands that are supported are #set, #echo and #include.

Syntax for embedded template commands

Embedded template 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 (hash), followed by the command. 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 code page 037, otherwise the command is ignored. The hexadecimal equivalents for these character sequences are X'4C5A60607B' and X'60606E'.

#echo command

The #echo command identifies a symbol that must be substituted when the template is inserted into the document. For example, this #echo command identifies the symbol ASYM to be substituted into a document at the location of the command:

```
This is the <!--#echo var=ASYM--> symbol.
```

When the template is used, the string containing the #echo command is completely replaced by the value defined for the symbol. If no symbol definition has been provided with that name, the #echo command remains in the output data. The value for the symbol can be defined by an application program, or as a default value by a #set command in the template.

#set command

The #set command is used to set the values of symbols. It is useful for setting up default values for symbols. For example, this #set command specifies a default value of 'first' for the symbol ASYM:

```
<!--#set var=ASYM value='first'-->
```

A #set commands can override another #set command. If you include more than one #set command in the template for the same symbol name, the last command is used.

A #set command in the template is ignored if the symbol to which it applies has already been given a value using the DOCUMENT SET command. A symbol which has been assigned a value using the DOCUMENT SET command can only be changed by issuing another DOCUMENT SET command.

The #set command can be used in combination with the #echo command, or it can apply to a symbol reference that has been specified by preceding the symbol name with an ampersand (&) and terminating it with a semicolon (;).

#include command

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-->
```

templatename is the name of the template (the 48 byte name) defined in the DOCTEMPLATE resource definition. The template name can also be enclosed in double quotes.

Chapter 40. Programming with documents and document templates

This section explains how to use documents and document templates in your application programs.

Creating a document

You can use the DOCUMENT CREATE command to create either an empty document, or a document containing data. The data can be a character string, a block of binary data, a document template, or a buffer of data.

About this task

The DOCUMENT CREATE command has a mandatory DOCTOKEN parameter requiring a 16–byte data-area. The document handler uses the DOCTOKEN operand to return a token, which is used to identify the document on subsequent calls.

To create a document containing data, you can specify options on the DOCUMENT CREATE command to:

- Include a character string (TEXT option).
- Include a block of binary data (BINARY option).
- Use a document template, specified by its template name (TEMPLATE option).
- Include the contents of a buffer of data (FROM option).
- Give values to any symbols in the document template or the item specified by the FROM option (SYMBOLLIST option).

These examples show the different ways you can create a document:

Procedure

1. To create an empty document and return its token, use the EXEC CICS DOCUMENT CREATE command with the DOCTOKEN option. This example creates an empty document, and returns the token in the 16-character variable MYDOC:

   ```
   EXEC CICS DOCUMENT CREATE
            DOCTOKEN(MYDOC)
   ```

2. Use the TEXT option to create a document that contains a character string specified by your application program. For example, if you define a character string variable called DOCTEXT and initialize 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(MYDOC1)
            TEXT(DOCTEXT)
            LENGTH(53)
   ```

   This string is added to the document unchanged, and CICS does not carry out any symbol substitution for it.

© Copyright IBM Corp. 1989, 2013

3. Use the BINARY option to create a document that contains binary data, which does not undergo code page conversion when the data is sent. This example creates a document consisting of the contents of a data-area as binary data:

```
EXEC CICS DOCUMENT CREATE
         DOCTOKEN(MYDOC2)
         BINARY(DATA-AREA)
```

   CICS does not carry out any symbol substitution for this data, and marks the data so that it is not converted to a client code page when you send the document to the recipient.

4. Use the TEMPLATE option to create a document using a document template that you have defined to CICS using a DOCTEMPLATE resource definition:
   a. Define a 48-byte variable, such as TEMPLATENAME, and initialize it to the value of the 48-character name of the template, as specified in the TEMPLATENAME attribute of its DOCTEMPLATE resource definition.
   b. If your document template contains no symbols, or you want to use the default values for the symbols, you can use the DOCUMENT CREATE command without the SYMBOLLIST option. For example:

```
EXEC CICS DOCUMENT CREATE
          DOCTOKEN(MYDOC3)
          TEMPLATE(TEMPLATENAME)
```

      It is important to note that you can only specify values for symbol substitution before, or at the time when, the document template is placed into the document. You cannot change the substituted values of the symbols after the template has been inserted.
   c. If you want to set values for symbols in the document template, use the DOCUMENT CREATE command with the SYMBOLLIST option. For example:

```
EXEC CICS DOCUMENT CREATE
          DOCTOKEN(MYDOC3)
          TEMPLATE(TEMPLATENAME)
          SYMBOLLIST('ORDER_NUMBER=0012345')
          LISTLENGTH(20)
```

5. Use the FROM option to create a document using a buffer of data. The buffer of data can contain symbol references that will be substituted in the same way as symbol references contained in document templates. For example:

```
EXEC CICS DOCUMENT CREATE
          DOCTOKEN(MYDOC4)
          FROM(BUFFER)
          SYMBOLLIST('ORDER_NUMBER=0012345')
          LENGTH(LEN)
```

Defining symbol values

Your application can define values for symbols in a document template using the DOCUMENT SET command or the DOCUMENT CREATE command. The symbol values are substituted when the template is used, either by a DOCUMENT CREATE command or by a DOCUMENT INSERT command.

About this task

You can define individual symbols using the SYMBOL and VALUE options on the DOCUMENT SET command. Alternatively, you can define several symbols in a single command using the SYMBOLLIST option on the DOCUMENT CREATE or

DOCUMENT SET command. By default, the symbol separator for the symbol list is an ampersand, but you can override this by using the DELIMITER option on the commands.

When you are designing your application, remember that:

- You must define values for the symbols before, or at the time when, you insert the document template into the document. You cannot change the substituted values of the symbols after the template has been inserted.
- Every symbol in a document template should be defined. If any symbol in the document template is not defined, either by the application program or by a #set command, no symbol substitution takes place for that symbol, and the finished document includes the symbol name or the #echo command specifying the symbol.
- A symbol definition provided by an application using the DOCUMENT SET command overrides a default value provided for the symbol by a #set command.
- A symbol which has been assigned a value using the DOCUMENT SET command can be changed by issuing another DOCUMENT SET command. If you do this after the document template has been inserted into the document, the new values are placed into the symbol table, and they 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.

"Rules for specifying symbols and symbol lists" on page 626 lists the rules relating to symbol names, special characters and spaces in symbol values, and symbol list separator characters.

Procedure

1. To define an individual symbol, use the DOCUMENT SET command 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. Follow the rules in "Rules for specifying symbols and symbol lists" on page 626. That topic explains the effects of the UNESCAPED option.
   a. Make sure you issue the DOCUMENT SET command **before** you issue the DOCUMENT INSERT command which places the document template in the document.
   b. You cannot use the DOCUMENT CREATE TEMPLATE command if you need to use DOCUMENT SET commands to set symbol values. Instead, you can create an empty document first, then define the symbol values with DOCUMENT SET commands, and then use the DOCUMENT INSERT command to insert the template that contains the symbol references.
2. To provide a symbol list containing multiple symbol definitions, use the SYMBOLLIST option on the DOCUMENT CREATE or DOCUMENT SET command. When you use TEMPLATE and SYMBOLLIST together on DOCUMENT CREATE, the symbols from the symbol list are resolved when the template is added to the document. If you are using the DOCUMENT SET command, make sure you issue it **before** you issue the DOCUMENT INSERT command which places the document template in the document.
   a. If the default symbol separator **&** (ampersand) is not suitable, because you want to use that character in a symbol value, use the DELIMITER option to specify an alternative character. Follow the rules in "Rules for specifying symbols and symbol lists" on page 626.

b. Use the SYMBOLLIST option to specify a buffer which contains the symbol list. Specify each symbol definition, separated by your chosen symbol separator. “Symbols and symbol lists” on page 606 shows an example of a symbol list. Follow the rules in “Rules for specifying symbols and symbol lists.” That topic explains the effects of the UNESCAPED option.

c. Use the LISTLENGTH option to specify the length of your symbol list. 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. If you do this:
   - The fixed list length that you choose should be long enough to include the maximum length of symbol list that you expect to supply.
   - When your fixed list length is greater than the actual length of the symbols that you supply, include an extra dummy symbol at the end of your symbol list, such as `'&END='`. This avoids the possibility of trailing spaces or unpredictable characters in the value of the last symbol in the list. Do not include this dummy 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.

Results

The symbols that you define are placed into the symbol table. If you are defining a symbol list on the DOCUMENT CREATE command, symbol substitution takes place immediately, and the document is created using the document template and your specified symbols. If you are using the DOCUMENT SET command to define individual symbols or a symbol list, the values you have placed in the symbol table are used for symbol substitution when a DOCUMENT CREATE or DOCUMENT INSERT command is issued using that document template.

Rules for specifying symbols and symbol lists

Each symbol has a name and a value. Follow these rules for selecting the symbol name and specifying values for symbols, either individually or in a symbol list.

Symbol names

The name of a 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.

To include a symbol in a document template, it can be specified as a symbol reference, which is the name of the symbol preceded with an ampersand (&) and terminated with a semicolon (;). Alternatively, the name of the symbol can be specified using the #echo command. When you specify the name of the symbol in your application, do not use an ampersand and semicolon. For example, the symbol reference &mytitle; in a template corresponds to the symbol name *mytitle* in a symbol list.

Separator in a symbol list

The SYMBOLLIST option on the DOCUMENT CREATE and DOCUMENT SET commands specifies 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 on the commands. Try to choose a symbol separator that will never be used within a symbol value in the symbol list, as special handling is required if you want to use the symbol separator within a symbol value. A non-printable character could be used.

There are several disallowed DELIMITER values. The disallowed values are:

- null (binary X'00')
- shift in (binary X'0E')
- shift out (binary X'0F')
- space (binary X'40')
- plus sign (binary X'4E')
- colon (binary X'7A')
- equals (binary X'7E')
- percent sign (binary X'6C')
- backslash (binary X'E0')

Special characters in symbol values

The values of symbols can contain any characters. However, special handling is required if you need to include the following characters in symbol values:

- The plus sign (+).
- The percent sign (%).
- The equals sign (=).
- The character that you have used as the symbol separator for a symbol list. Special handling is only required for this character when the symbol definition is provided in a symbol list, and does not apply when you use the DOCUMENT SET command to set an individual symbol value with the SYMBOL and VALUE options.

In the symbol value, you can use escape sequences to include characters such as these that have a special meaning. An escape sequence consists of the percent sign, followed by two characters that are hexadecimal digits (that is, 0–9, a-f, and A-F). When the symbol values are put into the symbol table, the percent sign and the two hexadecimal digits following it are replaced by the EBCDIC equivalent of the single ASCII character denoted by the two digits.

Some useful combinations are shown in Table 43.

Table 43. Escape sequences that can be used to represent special symbols

Character	Escape sequence
Plus sign +	%2B
Percent sign %	%25
Equals sign =	%3D
Ampersand & (default symbol separator)	%26

If the characters following the percent sign are not two valid hexadecimal digits, the percent sign and the following characters are put into the symbol table as they appear in the symbol list.

If you prefer not to use escape sequences, 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.

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.

Spaces in symbol values

If you want to include a space character in a symbol value, CICS allows you to use any of the following representations:

- A space character.
- A percent sign followed by the hexadecimal digits 20 (%20).
- A plus sign.

Any of these representations is interpreted as a space when the symbol value is put into the symbol table. This extends the HTML specification for the content type **application/x-www-form-urlencoded**, which uses a plus sign.

However, you **cannot** use a plus sign or the escape sequence %20 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.

Example: defining symbols without escape sequences

This example shows you how to pass symbol values to the document handler without using escape sequences. The symbol values in this list contain 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
```

In this example, the symbol COMPANY has a value of 'BLOGGS & SON', and the symbol ORDER has a value of 'NUTS+BOLTS'.

The use of a character other than the ampersand as the symbol separator means that an ampersand can be used in 'BLOGGS & SON'. 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 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 spaces are required in the symbol value 'BLOGGS & SON'. This means that the data no longer conforms to the specification for the content type **application/x-www-form-urlencoded**.

Adding more data to a document

When you have created a document, you can extend its contents by issuing one or more DOCUMENT INSERT commands. You can insert text, binary data, a buffer of data, a document template, or the value of a symbol. You can also insert bookmarks in a document, and use these to position later insertions.

About this task

You can specify options on the DOCUMENT INSERT command to:

- Insert a character string (TEXT option).
- Insert a block of binary data (BINARY option).
- Insert a document template, specified by its template name (TEMPLATE option).
- Insert the contents of a buffer of data (FROM option).
- Insert the value of a named symbol from the symbol table (SYMBOL option).

By default, the objects that you specify are added to the end of the document. To insert data in the middle of a document, you can set up one or more bookmarks. 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 placed between blocks of data - it cannot be placed in the middle of a block of data. You can use the DOCUMENT INSERT command to place a bookmark in a document during its construction, and then use the AT option to specify the bookmark when you insert a subsequent object. A special bookmark of 'TOP' is already defined, which you can use to insert data at the top of the document.

Procedure

1. Use the TEXT option to insert a character string specified by your application program. For example:

   ```
   EXEC CICS DOCUMENT INSERT
             DOCTOKEN(MYDOC)
             TEXT('Sample line 1.')
             LENGTH(15)
   ```

   This string is added to the document unchanged, and CICS does not carry out any symbol substitution for it.
2. Use the BINARY option to insert a block of binary data, which does not undergo code page conversion when the data is sent. For example:

   ```
   EXEC CICS DOCUMENT INSERT
            DOCTOKEN(MYDOC)
            BINARY(DATA-AREA)
   ```

   CICS does not carry out any symbol substitution for this data, and marks the data so that it is not converted to a client code page when you send the document to the recipient.
3. Use the TEMPLATE option to insert a document template. If you want to set values for the symbols in the document template, use the DOCUMENT SET command to specify individual symbols or a symbol list, **before** you issue the DOCUMENT INSERT command. “Defining symbol values” on page 624 explains how to do this.

4. Use the FROM option to insert a buffer of data. The buffer of data can contain symbol references that will be substituted in the same way as symbol references contained in document templates.
5. Use the SYMBOL option to insert the value of a symbol. SYMBOL specifies the name of a valid symbol whose value has been set in the symbol table (using the DOCUMENT SET or DOCUMENT CREATE command). The document handler inserts the value that is associated with the symbol into the document. Note that after you have inserted a value associated with a symbol into a document, you cannot change that value in the document that is being composed. If you subsequently 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.
6. Use the BOOKMARK option to insert a bookmark into the document. Bookmarks are placed during the construction of a document, and they provide an insertion point for the application to insert data that is obtained at a later stage. The name of a bookmark must be 16 characters long. It must not contain any imbedded spaces, and if your chosen name is less than 16 characters long, it must be padded on the right with blanks. For example, this sequence of commands creates a document with two blocks of text and a bookmark:

```
EXEC CICS DOCUMENT CREATE
          DOCTOKEN(MYDOCBOOK)
          TEXT('Pre-bookmark text. ')
          LENGTH(19)

EXEC CICS DOCUMENT INSERT
          DOCTOKEN(MYDOCBOOK)
          BOOKMARK('ABookmark       ')

EXEC CICS DOCUMENT INSERT
          DOCTOKEN(MYDOCBOOK)
          TEXT('Post-bookmark text. ')
          LENGTH(20)
```

7. Use the AT option with any of the other insertion options to place the object at a previously inserted bookmark, or at the special TOP bookmark. For example, this command inserts text at the example bookmark ABookmark:

```
EXEC CICS DOCUMENT INSERT
          DOCTOKEN(MYDOCBOOK)
          TEXT('Inserted at a bookmark. ')
          LENGTH(25)
          AT('ABookmark       ')
```

The completed document reads as follows:

```
Pre-bookmark text. Inserted at a bookmark. Post-bookmark text.
```

Replacing data in a document

You can place bookmarks in a document to delimit an area of data that can be replaced by a later insertion, or deleted. This technique lets you provide a default item of text or other data in your document, which can be used if your application finds that no data is available to replace it.

About this task

To set up and replace, or delete, a default item of data in a document:

Procedure

1. Create the document, specifying any data to be used at the beginning of the document. For this example, a document is created with some initial text, and its token is returned in the variable MYDOCREP:

```
EXEC CICS DOCUMENT CREATE
          DOCTOKEN(MYDOCREP)
          TEXT('Initial sample text. ')
          LENGTH(21)
```

2. Use the DOCUMENT INSERT command to specify the first bookmark:

```
EXEC CICS DOCUMENT INSERT
          DOCTOKEN(MYDOCREP)
          BOOKMARK('BMark1          ')
```

3. Use the DOCUMENT INSERT command to specify the item of text or other data to be replaced:

```
EXEC CICS DOCUMENT INSERT
          DOCTOKEN(MYDOCREP)
          TEXT('Text to be replaced. ')
          LENGTH(21)
```

4. Use the DOCUMENT INSERT command to specify the closing bookmark:

```
EXEC CICS DOCUMENT INSERT
          DOCTOKEN(MYDOCREP)
          BOOKMARK('BMark2          ')
```

5. Use the DOCUMENT INSERT command to add any data to be used at the end of the document:

```
EXEC CICS DOCUMENT INSERT
          DOCTOKEN(MYDOCREP)
          TEXT('Final sample text. ')
          LENGTH(19)
```

   At this point, the logical structure of this example document is:

```
Initial sample text. <BMark1>Text to be replaced. <BMark2>Final
sample text.
```

   The names of the bookmarks do not appear in the document.

6. To replace the text between the two bookmarks, BMark1 and BMark2, use the DOCUMENT INSERT command with the AT and TO options:

```
EXEC CICS DOCUMENT INSERT
          DOCTOKEN(ATOKEN)
          TEXT('Replacement Text. ')
          LENGTH(18)
          AT('BMark1          ')
          TO('BMark2          ')
```

   Now the example document appears as follows:

```
Initial sample text. Replacement Text. Final
sample text.
```

7. To delete the text between the two bookmarks, use the DOCUMENT INSERT command with the AT and TO options as above, but use the TEXT or BINARY option to specify a null string, with a LENGTH of zero.

Retrieving, storing and reusing a document

Documents created by an application exist only for the length of the CICS task in which they are created. To reuse a document, the application needs to retrieve a copy and save it.

Before you begin

This sequence of commands shows how a document can be created, retrieved, stored on a temporary storage queue, and reused as a document by the same or another application. To use this sequence of commands, the application program needs to define and initialize the following variables:

- 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.

About this task

Procedure

1. Create the initial document using the DOCUMENT CREATE command:

   ```
   EXEC CICS DOCUMENT CREATE
             DOCTOKEN(ATOKEN)
             TEXT('A sample document.')
             LENGTH(18)
   ```

   To help the application calculate the size of the buffer needed to hold a retrieved document, the document commands which alter the size of the document (the DOCUMENT CREATE and DOCUMENT INSERT commands) have a DOCSIZE option available. This value is the maximum size of the buffer needed to contain a copy of the document in its original codepage (including control information), when the RETRIEVE command is issued. However, when the CHARACTERSET option specifies an encoding that requires more bytes than the original EBCDIC data (for example, UTF-8), the maximum size might not be large enough to store the converted document. You can determine the actual document length before allocating the buffer by issuing a **DOCUMENT RETRIEVE** with a dummy buffer and a MAXLENGTH of zero, then handling the LENGERR condition and using the returned LENGTH value.
2. In the same task, issue the **DOCUMENT RETRIEVE** command to obtain a copy of the document in the application's own buffer.

   ```
   EXEC CICS DOCUMENT RETRIEVE
             DOCTOKEN(ATOKEN)
             INTO(DOCBUF)
             LENGTH(FWORDLEN)
             MAXLENGTH(20)
   ```

   By default, 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. CICS inserts tags into the document contents to identify the bookmarks and to delimit the blocks that do not require code page conversion. A document that you create from the retrieved copy is therefore identical to the original document. If you do not need to re-create the original document, you can modify the **DOCUMENT RETRIEVE** command as follows:

   a. To request a copy without control information, specify the DATAONLY option. With this option, CICS omits all the imbedded tags. The retrieved document contains no bookmarks, and there are no markings to delimit blocks that do not require code page conversion.

b. To convert the whole of the copy into a single client code page, specify the CHARACTERSET option.

3. Store the document on the temporary storage queue:

```
EXEC CICS WRITEQ TS
          QUEUE('AQUEUE')
          FROM(DOCBUF)
          LENGTH(HWORDLEN)
```

4. In the same or another application, read the stored data into the application's buffer:

```
EXEC CICS READQ TS
          QUEUE('AQUEUE')
          INTO(DOCBUF)
          LENGTH(HWORDLEN)
```

5. Use the **DOCUMENT CREATE** command with the FROM option to create a new document using the contents of the data buffer, that is, the retrieved document:

```
EXEC CICS DOCUMENT CREATE
          DOCTOKEN(ATOKEN)
          FROM(DOCBUF)
          LENGTH(FWORDLEN)
```

What to do next

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.

Deleting a document

You can use the DOCUMENT DELETE command to delete documents that are no longer required during a transaction. On execution of the command, the storage allocated to the document is released immediately. The DOCSTATUS(DOCDELETE) option of the WEB CONVERSE, WEB SEND (Client) and WEB SEND (Server) commands also allows document deletion.

About this task

DOCUMENT DELETE, WEB CONVERSE, WEB SEND (Client) and WEB SEND (Server) all use the DOCTOKEN to specify the 16–byte binary token of the document. The document token is returned when you create a document using the EXEC CICS DOCUMENT API commands.

To delete a document using the DOCUMENT DELETE command:

1. Specify the DOCTOKEN of the document you want to delete. For example:

```
EXEC CICS DOCUMENT DELETE
          DOCTOKEN(MYDOC)
```

2. The document is deleted from the document handling domain and storage is released immediately. If ACTION(EVENTUAL) is specified in the command, the Web domain retains a copy of the document.

WEB CONVERSE, WEB SEND (Client) and WEB SEND (Server) allow you to delete a document by specifying the DOCSTATUS(DOCDELETE) option. This option allows the application to indicate that it no longer requires the document once it has issued the CONVERSE or SEND command. The document is deleted from the document handling and Web domains on completion of the WEB SEND command, and storage is released immediately.

If you issue a WEB SEND, specifying DOCSTATUS(NODOCDELETE) and ACTION(EVENTUAL) in the command, it is possible to retrieve the document using the WEB RETRIEVE command. Using the DOCSTATUS(DOCDELETE) option or using the ACTION(IMMEDIATE) option will remove the document permanently from Web storage, and the document cannot be retrieved. The *CICS Application Programming Reference* provides more information on the limitations of document retrieval after deletion of documents.

Part 13. Named counter servers

Understanding and using named counter servers.

© Copyright IBM Corp. 1989, 2013

Chapter 41. 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 section describes:

- "The named counter fields"
- "Named counter pools" on page 638
- "Using the named counter EXEC interface" on page 640
- "Using the named counter CALL interface" on page 641
- "Named counter recovery" on page 653

The named counter fields

Each named counter consists of the counter name, the current value, the minimum value, and the maximum value.

© Copyright IBM Corp. 1989, 2013

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 640) 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 256 KB, 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 DFHNC*xxx* 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. 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.

For more information about EXEC CICS COUNTER and DCOUNTER commands see "Using the named counter EXEC interface" on page 640.

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 Setting up shared data sets, CSD and SYSIN in Configuring. This also describes the DFHNCO macro that you can use to create your own options table.

The POOLSEL parameter in the default options table works 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 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 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 DFHNC*xxx*, or
- Your CICS region uses only one pool that can be defined by the NCPLDFT system initialization parameter.

Note:

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.

You can use EXEC CICS commands 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 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 then 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 must 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 excuting 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 you need to consider the following in your application program.

About this task

1. Ensure that 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 DFHNC*xxx* 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 that 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 CICSTS51.CICS.SDFHLINK.

CICS provides copybooks for all the supported languages:

Assembler
The standard assembler named counter interface definitions are provided in copybook DFHNCASM. Include these in your application programs using COPY DFHNCASM 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,CTRLEN,CTR),VL
```

C/C++
The named counter interface definitions for C/C++ are provided in header file DFHNCC. The symbolic constant names are in uppercase. The function name is `dfhnctr`, in lowercase.

COBOL
The named counter interface definitions for COBOL are provided in copybook DFHNCCOB.

Earlier versions of COBOL that are supported by CICS do not allow underscores in names. The symbolic names provided in copy book

DFHNCCOB therefore 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

This diagram shows you the syntax of the named counter DFHNCTR call.

:

```
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 135. DFHNCTR call syntax-PL/I illustration

Note:

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 embedded 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 650.
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 ((*maximum_value* plus 1) minus *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 651 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 638 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 649.

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 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 remains 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
function	01 FUNCTION	PIC S9(8) COMP VALUE +1.
return_code	01 NC-RETURN-CODE.	PIC S9(8) COMP VALUE +0.
pool_selector	01 NC-POOL-SELECTOR	PIC X(8).
counter_name	01 NC-COUNTER-NAME	PIC X(16).
value_length	01 NC_VALUE-LENGTH	PIC S9(8) COMP VALUE +4.
current_value	01 NC-CURRENT-VALUE	PIC S9(8) VALUE +0.
minimum_value	01 NC-MIN-VALUE	PIC S9(8) VALUE +0.
maximum_value	01 NC-MAX-VALUE	PIC S9(8) VALUE -1.
counter_options	01 NC-OPTIONS	PIC S9(8) COMP VALUE +0.
update_value	01 NC-UPDATE-VALUE	PIC S9(8) VALUE +1.
compare_min	01 NC-COMP-MIN	PIC S9(8) VALUE +0.
compare_max	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 previously described, 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

The named counter call interface has three warning return codes (1 - 3), 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 following list, 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 can 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, that a program might not to 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 application 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 cannot establish a connection to the server for the appropriate named counter pool. Further information can be found in an AXM services message in the application 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 aplication job log and the server job log.

307 (NC_NAME_TOKEN_ERROR)
An IEANT*xx* name/token service call in 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, cannot 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. For example, one reason for this situation 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 retries until it succeeds, with the result that the application program has 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 re-create 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, if 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, re-create it with an appropriate value that you have reconstructed from other information, using the following methods described.
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 re-create 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 re-create 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 re-create 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 careful verification and testing, because the logic to handle duplicate keys is normally only exercised in unusual recovery situations.

If it is difficult to re-create 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 re-create 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 higher, 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 14. Printing and spool files

Understanding and using CICS facilities for printer control and spool files.

© Copyright IBM Corp. 1989, 2013

Chapter 42. Printing and spool files

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.

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.

Formatting for CICS printers

The application programming interface for writing to a printer terminal is essentially the same as for writing to a display. 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).

The problem of arranging that your task has the printer as its principal facility is discussed in "Using CICS printers" on page 667.

"BMS support levels" on page 687 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 Chapter 30, "Using batch data interchange," on page 545.

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.

© Copyright IBM Corp. 1989, 2013

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.

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.

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 551 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 665.

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 556. 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 example” on page 559. If you send this same data stream to a 3270 printer, it prints an image of the screen shown in Figure 128 on page 559. 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 “NLEOM option” on page 663 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 “Outbound data stream example” on page 559, which uses buffer control orders.

Table 44. 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, HONEOM, NLEOM, FORMFEED, and 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 3270 Data Stream Device Guide and the IBM 3270 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 a 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 “The output datastream” on page 551. 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 at “NLEOM option” on page 663.

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 (options L40, L64 and L80). 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 printed, as graphics.

If you use L40, L64, or L80 under BMS, only use the value that corresponds to the page width in your terminal definition (see "Determining the characteristics of a CICS printer" on page 666). 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 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 666 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, if 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. 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.

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 IBM 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 is mistaken for a set attribute control order, in which case the two bytes following it are 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 might be recognized and altered by BMS, the recommended method is to use a terminal control SEND command, rather than BMS.

SCS input

SCS printers have limited input capability, in the form of *program attention* keys.

However, these keys are not like the attention keys that are described in “Input from a 3270 terminal” on page 562. 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.

Using printers with CICS

You can use the following to show you how to program for CICS printers and non-CICS printers.

Determining the characteristics of a CICS printer

If you are writing a program that supports more than one type of CICS printer, you might need to determine the characteristics of a particular printer. You can use the ASSIGN and INQUIRE TERMINAL commands for this purpose.

The ASSIGN options that apply to terminals, including several that are specific to printers are shown in Table 37 on page 535.

The INQUIRE TERMINAL options that apply specifically to printers and the corresponding parameters in the terminal definition are shown in Table 45:

Table 45. 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)
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 theIntroduction to System programming commands in Developing system programs 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. The following table lists the values used.

Table 46. 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 SCRNHT and SCRNWD 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.
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.

Related concepts:

"Device-dependent maps" on page 705
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.

Using CICS printers

You can print to a CICS printer using the following methods; printing with a START command, printing with transient data, or printing with BMS routing.

About this task

As we noted, 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 using one of the following methods.

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.

About this task

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 136. 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 137. 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.

About this task

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 typically 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)::

This example shows you the syntax for writing data to a transient data queue called 'PRT1' which is the predefined symbolic destination for 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::

This example shows you how 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.

```
   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)
```

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 599 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 36, "Transient data control," on page 597 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.

About this task

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 "Message routing" on page 750 for more information about routing.)

Using Non-CICS printers

The steps listed here show you how to format your output, deliver your output, and notify the application when output is ready for printing, on a printer managed outside CICS:

Procedure

1. Format your output in the manner required by the application or subsystem that controls the printer you want 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 the 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. However, for most printers you need to meet the format requirements of the application that drives the printer.

Formatting output with BMS, is explained in "Programming for non-CICS printers" on page 671. The format requirements of the application that drives the printer can be the device format or an intermediate form dictated by the application. For conventional line printers, formatting is a matter of producing line images and, sometimes, adding carriage-control characters.

Non-CICS printers: Delivering the data

Print data is typically 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 typically one of the file types listed in the first column of Table 47. The second column of the table shows which groups of CICS commands you can use to create such data.

Table 47. 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)
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 can be:

- An extra-partition transient data queue (see Chapter 36, "Transient data control," on page 597 for information about transient data queues)
- The output half of a sequential terminal (see Chapter 29, "Using sequential terminal support," on page 543 and "Support for non-3270 terminals" on page 703)
- A spool file (see "CICS interface to JES" on page 674)

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 668. 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 later.) 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 typically large, but it is possible for a job that runs a 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 large number of spool files, consult your system 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 47 on page 670, 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 to copy a display screen to a printer, you have choices additional to the ones 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. For more information, see TERMINAL resources in Reference -> System definition.

CICS print key

The first such option is the *CICS print key* (also called the *local copy key*). This option 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 DISPLAY and PRINT options for combined lists of information in Reference -> System definition.

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 z/OS Communications Server 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 to 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 z/OS Communications Server 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 typically 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.

You can use the **`ISSUE COPY`** command 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. It

copies the buffer of the terminal named in the TERMID 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; for the bit settings in this control character, see IBM 3270 Data Stream Device Guide. 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 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. For information about CSPG, see CSPG - page retrieval in Reference > System definition.

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 676). 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 138

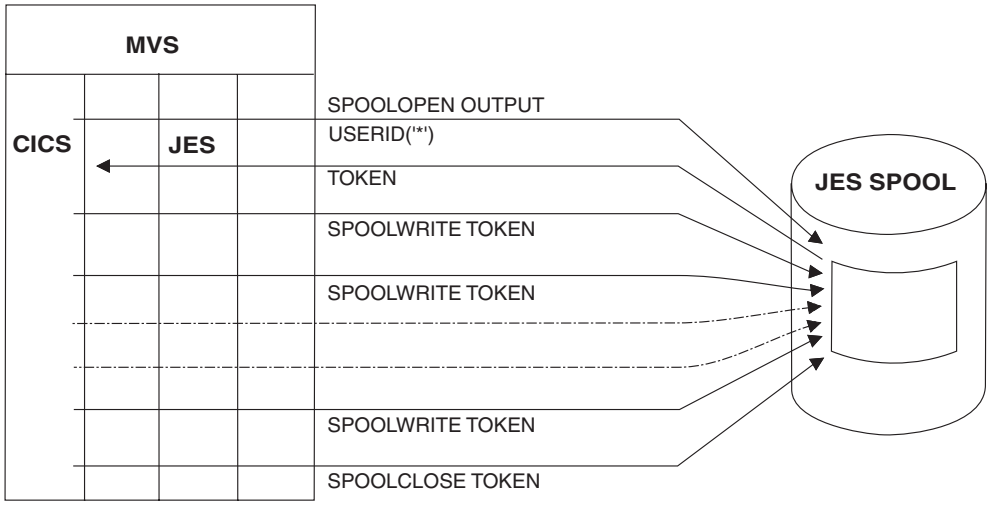


Figure 138. Create a file and write it to the JES spool

- 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 139

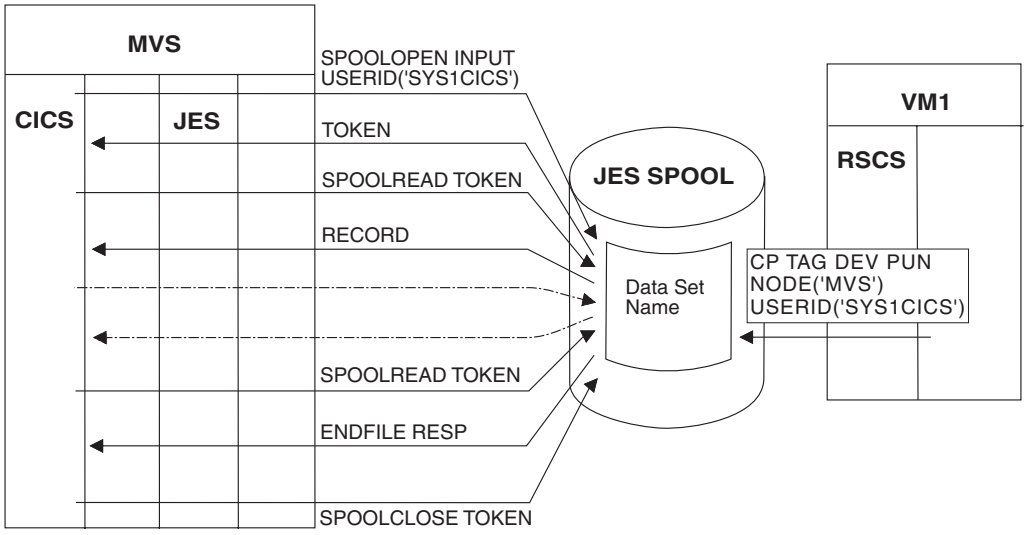


Figure 139. Retrieve data from the JES spool

- 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 140.

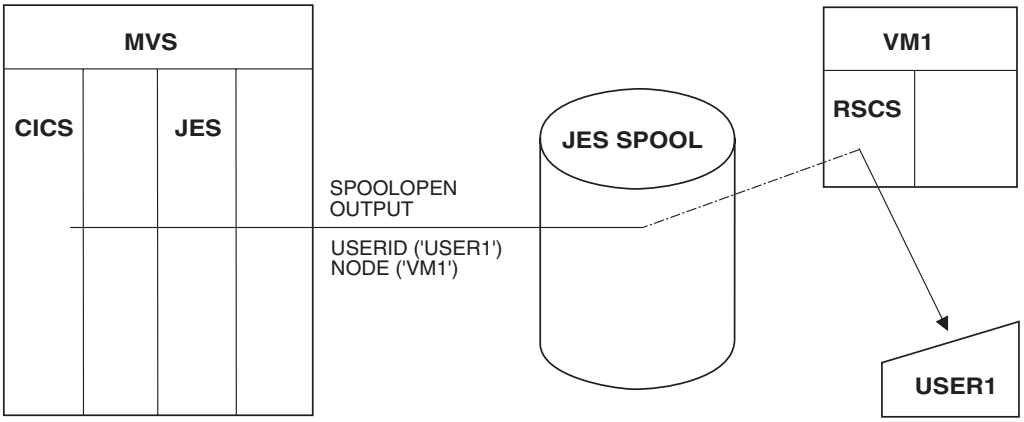


Figure 140. Send a written file to a remote destination

Using the CICS interface to JES

To use the CICS interface to JES, you must define the DFHSIT SPOOL=YES system intialization parameter in your CICS startup JCL.

About this task

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 must 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 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 large files or files that can 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, the program starts by issuing a SPOOLOPEN OUTPUT command, to allocate an output data set. Records are added to the file using the SPOOLWRITE command after which the SPOOLCLOSE command is used to release the file to JES for delivery or processing.

About this task

The NODE and USERID options on the SPOOLOPEN OUTPUT 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 679 for more information about the naming of spool files.

Reading input spool files

The command sequence for reading a spool file is like 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.

About this task

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.

Like 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 has not done 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; 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 139 on page 675 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 typically 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) knows what name the sender used, and that no other applications in the operating system of the receiver uses 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 matches its own z/OS Communications Server 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 z/OS Communications Server in the MVS system in which your CICS is executing).

The meaning of USERID varies with the operating system. In VM, it is a specific user; in MVS, it might 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 141. 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
              .
              .
              .
```

Note:

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 area obtained by using the GETMAIN request is then moved to the first word pointed to by L-FILLER being L-ADDRESS (thus 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. Therefore W-POINTER points to an area that has a pointer pointing to the OUTDESCR data.

Examples of SPOOL commands

The following examples show you SPOOL commands using COBOL, PL/I, C, and ASSEMBLER.

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)
```

Part 15. Basic Mapping Support (BMS)

Using basic mapping support to control terminal devices.

© Copyright IBM Corp. 1989, 2013

Chapter 43. 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 Terminal control in Developing applications.

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. Allowing you to use the same program for various 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.

Related tasks:

"Sending BMS mapped output" on page 708
Before you can send BMS mapped output, you must perform the following steps to move the output data into the map structure.

Related information:

"Creating the map" on page 691
BMS provides three assembler language macro instructions (**macros**) for defining maps. This method of map definition is still widely used, and this is the method that is described here.

"Using the SEND MAP command" on page 713
This list shows you what the SEND MAP command tells BMS.

"Receiving mapped data" on page 722
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.

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 of BMS. If your installation uses minimum or standard BMS, note the features that require levels beyond yours.

The features 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.

© Copyright IBM Corp. 1989, 2013

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 "fast path" on commands that 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)
- 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. BMS is fully supported on CICS Transaction Server for z/OS and CICS Transaction Server for VSE. However, there are differences among platforms even at the same level, typically imposed by differences in the runtime environment. These are described in detail, by function, in *CICS Family: API Structure*. If your application might eventually move to another platform, or there is a chance that the graphical user interface part of it might get distributed to one, you must consult that manual.

A BMS output example

This example shows how BMS creates a formatted screen. It takes a list of data items from a program and displays them on the screen according to a predefined format.

BMS creates a formatted screen by merging 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 section is concerned with display screens, but most of it applies equally to printers. Chapter 42, "Printing and spool files," on page 659 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 703.

You define the formats, called *maps*, separately from the programs that use them. Allowing you to reposition fields, change their attributes, and change the constant text without modifying your programs. If you add or remove variable data, 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, and produces the screen shown in Figure 142 in response:

```
QCK                     Quick Customer Account Check
Account:    0000005
Name:       Thompson        Chris
Max charge: $500.00
```

Figure 142. Normal "quick check" output screen

The program uses the entered account number to retrieve the customer's record from the account file. From the information in this record, it completes 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 143:
This message is to be highlighted, to draw the clerk's attention to it.

```
QCK                     Quick Customer Account Check
Account:    0000005
Name:       Thompson        Chris
Max charge:    $0.00
STOLEN CARD - SECURITY NOTIFIED
```

Figure 143. "Quick check" output screen with warning message

You must first define the screen. We explain how to do so for this particular map in "Creating the map" on page 691. For the moment, however, let us assume that one of the outputs of this process is a data structure like the one in Figure 144. (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  QCKMAPO.
    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 144. 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 711 and "Receiving mapped data" on page 722.

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 144 on page 690) 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 ACCTNOO.
      MOVE ACCTFILE-SURNAME TO SURNAMEO.
      MOVE ACCTFILE-FNAME TO FNAMEO.
      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 DFHBMBRY 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 145. BMS output example

Creating the map

BMS provides three assembler language macro instructions (**macros**) for defining maps. This method of map definition is still widely used, and this is the method that is described here.

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 it goes on to describe how to get 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 32, "The 3270 family of terminals," on page 551 for information on 3270 terms.

Related concepts:

Chapter 43, "Basic mapping support," on page 687
Basic mapping support (BMS) is an application programming interface between CICS programs and terminal devices.

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.

About this task

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 can 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 lowercase), $, #, @, numeric digits 0 - 9, and the underscore (_) character.

There is one exception to this rule. The hyphen (-) character can be used in field names if:

- The map set is only used by application programs written in COBOL.
- The map set 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 previously.

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 (in earlier versions) an underscore. 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 can therefore be up to 31 characters in length. The assembler, PL/I, 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*.

Figure 146 shows the field definitions for the map we considered in Figure 143 on page 690.

```
        DFHMDF POS=(1,1),LENGTH=3,ATTRB=(ASKIP,BRT),INITIAL='QCK'
        DFHMDF POS=(1,26),LENGTH=28,ATTRB=(ASKIP,NORM),                X
              INITIAL='Quick Customer Account Check'
        DFHMDF POS=(3,1),LENGTH=8,ATTRB=(ASKIP,NORM),INITIAL='Account:'
ACCTNO  DFHMDF POS=(3,13),LENGTH=7,ATTRB=(ASKIP,NORM)
        DFHMDF POS=(4,1),LENGTH=5,ATTRB=(ASKIP,NORM),INITIAL='Name:'
SURNAME DFHMDF POS=(4,13),LENGTH=15,ATTRB=(ASKIP,NORM)
FNAME   DFHMDF POS=(4,30),LENGTH=10,ATTRB=(ASKIP,NORM)
        DFHMDF POS=(5,1),LENGTH=11,ATTRB=(ASKIP,NORM),INITIAL='Max charge:'
CHG     DFHMDF POS=(5,13),ATTRB=(ASKIP,NORM),PICOUT='$,$$0.00'
MSG     DFHMDF LENGTH=20,POS=(7,1),ATTRB=(ASKIP,NORM)
```

Figure 146. 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 in "Building the output screen" on page 716 for further information.)
3. The **ATTRB** (attributes) option sets the **field attributes** of the field, which we discussed in "3270 field attributes" on page 554. 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 711.
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 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 144 on page 690 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 700 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.

About this task

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 DFHMDF 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 705)

The map name and size are the critical information about 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 create a map using the DFHMSD macro which defines a map set

About this task

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 699.) 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 697)
- 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 144 on page 690). 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 "Sending BMS mapped output" on page 708.

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:

```
         DFHMSD TYPE=FINAL
```

Writing BMS macros

Because a BMS macro is an assembler language statement, you must follow assembler language syntax rules.

The following set of rules work, but they are more restrictive than the actual rules. For the full set of assembler language syntax rules, see *Assembler H Version 2 Application Programming Language Reference*.

1. Start names in column 1. Map and map set names can 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 one-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 709 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, DFHMDI; 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 might 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 uppercase 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 previous examples.) 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   DFHMDF POS=(02,21),LENGTH=20,PS=8,GINIT='<D1D2D3D4D5D6>******
                <D7D8D9D0>'
  IXMPL1   DFHMDF 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 on page 696.
- 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.

About this task

You typically 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

You can use the TYPE=MAP assembly, followed by a link-edit to produce a load module called the physical map set. The TYPE=DSECT assembly is used to produce a series of data structures, collectively called the symbolic 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, 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 144 on page 690) and used it in the example code. Symbolic map sets are typically stored in the library your installation defines for source code that gets copied into programs. Member names are typically 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 cataloged 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 with map assemblies, and Chapter 10, “Installing map sets and partition sets,” on page 149 for more information about assembling maps.

Note:

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 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 can want to change by program. It is always safest to recompile.

The SDF II alternative

If you use the IBM licensed program Screen Definition Facility II the assembly and link-edit steps are not required. SDF II 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 710. 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 5 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=ADS|ADSL 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 WebSphere MQ.

Map sets generated with CICS releases before CICS Transaction Server for z/OS, Version 5 Release 1 do not contain the ADS descriptor.

The format of the ADS descriptor is contained in the following copybooks:

Table 48. 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 5 Release 1 to re-create BMS macro source from a mapset load module.

See BMS macro generation utility program (DFHBMSUP) in the *CICS Operations and Utilities Guide*for more information about DFHBMSUP.

Using complex fields

BMS provides two options to the DFHMDF macro for defining complex fields; GRPNAME and OCCURS.

About this task

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 144 on page 690). 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

The GRPNAME option is used to generate symbolic storage definitions and to combine specific fields under one group name. The same group name must be specified for each field that is to belong to the group.

About this task

Sometimes, you have to refer to subfields within a single field on the display. For example, you can have a date field that appears on the screen as follows:

```
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 previous date 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

The OCCURS option specifies that the indicated number of entries for the field are to be generated in a map, and that the map definition is to be generated in such a way that the fields are addressable as entries in a matrix or an array.

About this task

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 DFHBMBRY to TELNOA (I).
```

(DFHBMBRY is a CICS-supplied constant for setting the field intensity to bright; we explain more in "Attribute value definitions: DFHBMSCA" on page 713.)

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 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 the same field structure as the original. 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 can 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.

The symbolic output map 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 143 on page 690 would look like this in block data format (assuming a map 80 columns wide). Compare this with the normal "field data". format (in Figure 144 on page 690)

from the same map definition.
You can set only the field attributes in the program; BMS ignores the DSATTS

```
01  QCKMAPO.
    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 147. 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 can 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

Support for non-3270 terminals requires standard BMS; minimum BMS supports only 3270 displays and printers.

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 49 on page 707. See "Non-3270 CICS printers" on page 665 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 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 than on output 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 remained 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 options 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 Chapter 24, "Using data transmission commands," on page 527.
- 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 775 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 663.

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.

Device-dependent maps are 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 148. "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 149. 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 722, 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

Device dependent support (DDS) is an installation feature that enables device-dependent maps.

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 terminal. You also can use JCL or the link-edit NAME statement to control the member name under which a map set is stored. 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.

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 49.

Table 49. Terminal codes for BMS

Code	Terminal or logical unit
A	CRLP (card reader input, line printer output)
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 previously described. 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 the options that apply are described in Chapter 27, “Finding out about your terminal,” on page 535.

There are also ASSIGN options specific to BMS, but you need them most often when you use the ACCUM option, and so these are described later, in “ASSIGN options for cumulative processing” on page 743.

Sending BMS mapped output

Before you can send BMS mapped output, you must perform the following steps to move the output data into the map structure.

About this task

When you have assembled your symbolic map set, you are ready to code. The example used in “Creating the map” on page 691 describes how you get data from an application program to a map. This process is discussed 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:

Procedure

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.

What to do next

Related concepts:

Chapter 43, "Basic mapping support," on page 687
Basic mapping support (BMS) is an application programming interface between CICS programs and terminal devices.

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.

About this task

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 245.) 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 "Storage control" on page 579.) 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 in Figure 145 on page 691, but we could have used a GETMAIN. If we had, the code 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 previous example. 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 that remains there by a previous process is not used inadvertently.

About this task

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 724, and using the same map you used for input is discussed in "Sending mapped output after mapped input" on page 732.

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 named the map suffixed by the letter *O*. You can see this in Figure 144 on page 690, 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 might be easier to clear the map one field at a time, as you edit the input (see "Handling input errors" on page 730).

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. Firstly, we explain to you about the data, and then we explain to about the attributes.

In the typical 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 in A BMS output example 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 715 and summarize the exact rules in "Building the output screen" on page 716.

Setting the display characteristics

Display attributes are the second component of the output data. For named fields, BMS generates subfields which can be used to override the map assigned value in your program.

In A BMS output example, 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. (See "3270 field attributes" on page 554 for information about attributes).

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 50 on page 712. 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 MAPATTS option (even if these attribute types do not appear in any DFHMDF macro).

Table 50. 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 Chapter 24, "Using data transmission commands," on page 527). 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 Chapter 27, "Finding out about your terminal," on page 535. There are also provisions in BMS for keeping your program independent of the terminal type; see "Device-dependent maps" on page 705.

Changing the attributes

Field attributes in map definitions can be changed using the options MAPATTS and DSATTS.

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 flash. 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 144 on page 690 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 flashing 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 in "Building the output screen" on page 716.

Attribute value definitions: DFHBMSCA

The 1 byte values required to set attribute values are bit combinations defined by 3270 hardware. CICS provides source code, named DFHBMSCA, which defines the commonly used values for all attributes, and assigns meaningful names to each combination.

You can copy DFHBMSCA into your program. The program DFHBLINK in "Changing the attributes" on page 712 is an example. To define DFHBLINK, copy DFHBMSCA into the working storage, as follows:

```
      WORKING-STORAGE SECTION.
      ...
      01  COPY DFHBMSCA.
```

A version of DFHBMSCA is provided for each programming language. The value names are the same in all versions. If you need an attribute combination that is not included in DFHBMSCA, to determine the value, see *3270 Data Stream Programmer's Reference*. If you use the value often, you can modify DFHBMSCA to include it.

In assembler language only, the values are defined with EQUates, so you use MVI rather than MVC instructions.

Using the SEND MAP command

This list shows you what the SEND MAP command tells BMS.

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 720.

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.

Related concepts:

Chapter 43, "Basic mapping support," on page 687
Basic mapping support (BMS) is an application programming interface between CICS programs and terminal devices.

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 in "The output datastream" on page 551 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 in "Merging the symbolic and physical maps" on page 715.

  FRSET (field reset) turns off the modified data tag of all fields in the buffer; "3270 field attributes" on page 554 explains more about this option.
- **FREEKB** (free keyboard) unlocks the keyboard when the output is sent to the terminal. You typically 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 661. **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 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 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 z/OS Communications Server 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 remain 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 714 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 730.)

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 in "The output

datastream" on page 551. 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 in the same state it was in 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 can 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.

Within the map area, what is sent depends on whether the DATAONLY option is present. In the typical 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) remain unchanged. (There might or might 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, 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 can come from four sources: the program; the map; the hardware's defaults; or the previous contents of the screen.

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 50 on page 712). 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 remain 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 previous conditions 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

Your map need not be the same size as your screen or printer page. Your application can use only a part of the screen area, or you might 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 “BMS logical messages” on page 734. 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 might produce a momentary “flash” at a display device. For this reason, and to eliminate the path length of extra I/O, you might prefer to use logical messages for such composite screens.

Outside the map 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 can 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 might change, even though the contents do not.

Using GDDM and BMS

One use of the buffer overlay technique 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 might be using.

Positioning the cursor

Positioning the cursor is important when you use a map for input. 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.

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 726, and about INOUT maps in "Receiving mapped data" on page 722.) 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 729 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

Most of the exception conditions that can occur on SEND MAP and SEND CONTROL commands 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, such as color, to a terminal that is not defined to support that attribute.

For more information about the error conditions that can occur on SEND MAP and SEND CONTROL commands, see CICS API commands in Reference -> Application development.

The effects of invalid data depend on both the specific terminal and the nature of the incorrect data:

- Invalid data might be interpreted as a control sequence, so that the device accepts the data but produces the wrong output.
- The screen might display an error indicator.
- An ATNI abend might occur. The point at which your task is notified of an ATNI depends on whether you specified the WAIT option (see "SEND MAP control options" on page 714).

Output disposition options: TERMINAL, SET, and PAGING

The output disposition options specify what BMS should do with the formatted output stream. 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; SET and PAGING.

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 709 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 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 "Message routing" on page 750, 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 758.

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 51. Page list entry format

Bytes	Contents
0	Terminal type (see Table 49 on page 707)
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 52. 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.

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.

Related concepts:

Chapter 43, “Basic mapping support,” on page 687
Basic mapping support (BMS) is an application programming interface between CICS programs and terminal devices.

An input-output example

This example shows you the map definition for a simple data entry program called "quick update".

Before we explain the details of the input structure, let us reexamine the “quick check” example in “A BMS output example” on page 689. 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 about the screen, so

```
QUP                     Quick Account Update
Current charge okay; enter next
Account:   _______
Charge:   $ _______
```

Figure 150. Normal “quick update” response

that 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 151. "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 like those

```
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 152. Map definition for input-output map

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 the *CICS Application Programming Reference* for details on using PICIN.

Figure 153 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 153. 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 142 on page 689 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 located 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 142 on page 689 and Figure 153 on page 724.

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 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.

If the transaction also calls for mapped output, as “quick update” and most other transactions do, you continue with the steps outlined before, in “Sending BMS mapped output” on page 708. Some considerations and shortcuts for mapped input are described in “Sending mapped output after mapped input” on page 732.

Using the RECEIVE MAP command

The RECEIVE MAP command causes BMS to format terminal input data and make it accessible to your application program.

The command specifies:

- 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 uppercase (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 734).

BMS also translates lowercase input to uppercase automatically in some cases; we explain how to control translation in "Uppercase translation" on page 728.

You tell BMS where to put the formatted input with the INTO or SET option of RECEIVE MAP.

In addition to the data on the screen, the RECEIVE MAP command tells you where the operator last used 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 last located.

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 **GETMAIN** command.

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 709 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 previously shown. You can specify some other variable if you want, 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 "Storage control" on page 579).

Formatted screen input

CICS normally reads a 3270 screen with a *read modified* command. CICS provides an option, BUFFER, for the terminal control RECEIVE command, with which you can capture the entire contents of a 3270 screen.

The data transmitted depends on what the operator did to cause transmission:

- The ENTER key or a function 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; how to do this is explained in "Using the attention identifier" on page 728. You can also find more detail on 3270 input operations in "Input from a 3270 terminal" on page 562.

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 in "MAPFAIL and other exception conditions" on page 733. Field selection features transmit field data, but in most cases not the same data as the ENTER and function keys, which we describe in the paragraphs that follow. See "Support for special hardware" on page 770 for the exceptions if you plan to use these features.

Most applications are designed for transmission by the ENTER key or a function 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.

Related concepts:

"Reading from a 3270 terminal" on page 563
There are two basic read commands for the 3270, read buffer and read modified.

"Support for non-3270 terminals" on page 703
Support for non-3270 terminals requires standard BMS; minimum BMS supports only 3270 displays and printers.

Modified data

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.

Modification is explained in "3270 field attributes" on page 554. 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 729 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 can 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.

Uppercase translation

CICS converts lower case input characters to uppercase 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 resource definitions in the *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 225.) 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 734).

Using the attention identifier

The attention identifier 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 = DFHPF12,
    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 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 713). The contents of the DFHAID copy book are listed in the *CICS Application Programming Reference* manual.

Using the HANDLE AID command

You can use a HANDLE AID command to identify the attention key used.

HANDLE AID is used to specify the label to which control is to be passed when an attention identifier (AID) is received from a display device.

Restriction: This command is supported only in COBOL, PL/I, and assembler language applications (but not AMODE(64) assembler language applications). It is not supported in all other supported high level languages.

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.

The following example shows an alternative to the “escape” code:

```
       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 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 exception condition that occurs and whether a HANDLE CONDITION is active for that exception. Therefore, HANDLE AID can mask an exception condition if you check for it with HANDLE CONDITION. It can be preferable to use an alternative test for the AID, the exceptional conditions, or both. You can check EIBAID for the AID, and use the RESP option or check EIBRESP for exceptions. This situation is especially significant for the MAPFAIL condition; see “MAPFAIL and other exception conditions” on page 733.

Finding the cursor

In some applications, you need to know where the operator last located 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 last located. This only works, of course, if the cursor is located 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 BMS macros in Reference -> Application development 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

This example shows you how the input subfields are used, using the "quick update" program.

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 719. 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 DFHUNIMD 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 DFHBMBRY 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

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 must 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 DFHBMBRY). 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

When the operator enters correct and incorrect data, the correct data should be saved, so that the operator does not have to rekey anything except corrections. 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 "Sharing data across transactions" on page 261 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 must 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 must 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 later 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 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 about the screen.

Rechecking

The last requirement is to ensure that the input data is rechecked. If your task is conversational, this means repeating the audit section of your code after you have received (and merged, if necessary) the corrected input. In a pseudoconversational sequence, you typically 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 must 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

After a transaction has processed its initial input, the next step is to prepare and send the transaction output.

In general, if the output is to be mapped, you follow the steps outlined in "Sending BMS mapped output" on page 708. 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 710 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 715 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 exception conditions

The MAPFAIL exception condition occurs when no usable data is transmitted from the terminal, or when the data transmitted is unformatted.

The MAPFAIL condition 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 when both the following conditions apply:

- 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).
- The cursor was not located in a field defined in the map and named, or the map did not specify CURSLOC=YES.

An operator might easily press ENTER too soon, or a “short read” key accidentally. To be user friendly, you might want to refresh the screen after a MAPFAIL condition, rather than ending the transaction in error.

A MAPFAIL condition 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 that is different from the one you sent. This situation might signal an error in logic, or it might mean that your transaction is in its startup phase. For example, the quick update example does not allow for getting started; that is, for getting an empty map onto the screen so that the operator can start to use the transaction. You could use a separate transaction to do this, but you can take advantage of the code you need to refresh the screen after a MAPFAIL. The required code is as follows:

```
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.
```

This code reminds the operator how to escape, because attempts to do this might be the original cause of the MAPFAIL. If you do not want to send this message, or if it was the default in the map, you can use 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 you must 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 for other exception conditions. However, if you issue a HANDLE CONDITION command and you also have a HANDLE AID active for the AID you receive, control goes to the label specified for the AID and not that for MAPFAIL. For further explanation, see "Using the HANDLE AID command" on page 728. In this situation, you will not be aware 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 z/OS Communications Server. 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 uppercase with the ASIS option when you use FROM. Moreover, BMS does not set EIBAID 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.

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 want. Details on cumulative page building are in "Cumulative output — the ACCUM option" on page 739.

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 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.

Building logical messages

When you construct a logical message, you must 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 want.
- 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 previously noted 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 746.) 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 "Partition support" on page 763 and "Logical device components" on page 770 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 749 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 later in the same task, if you want.

RETAIN and RELEASE

Options on the **SEND PAGE** command tell BMS when and how to deliver the pages to the terminal.

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 738).

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

Options on the **SEND PAGE** command tell 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. You 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 previously noted. 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 CSPG - page retrieval in the *CICS Supplied Transactions* 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 Troubleshooting for recovery processing in Troubleshooting and support.

Routed messages are eligible for recovery, as well as messages created for your principal facility. Routing is described in "Message routing" on page 750.

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 want, 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.)

Floating maps: how BMS places maps using ACCUM

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 feature is 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 want 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 for 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 on page 743), 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 on page 742 (the LINE option is ignored if JUSTIFY=LAST); if not, overflow occurs (Step 5 on page 743).

      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*).
   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 on page 743.

      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 on page 743).

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 the right, and COLUMN=SAME is the right-hand column of the map most recently placed which had JUSTIFY=RIGHT and was not a header or trailer.

   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 on page 741) after each adjustment of the starting line.

4. If the map fits, BMS adds it to the current page and updates the available space, using the following rules:
   - Lines above the first line of the map are completely unavailable.
   - If the map specifies JUSTIFY=LEFT, the columns from the left edge of the page through the right-most column of the map are unavailable on the lines from the top of the map through the last line on the page that has anything on it (whether from this map or an earlier one).
   - If the map specifies JUSTIFY=RIGHT, the columns between the right-hand edge of the page and the left-hand edge of the map are unavailable on the lines from the top of the map through the last line of the page that has anything on it.

   Figure 154 on page 743 shows how the remaining space is reduced with each new map placed.

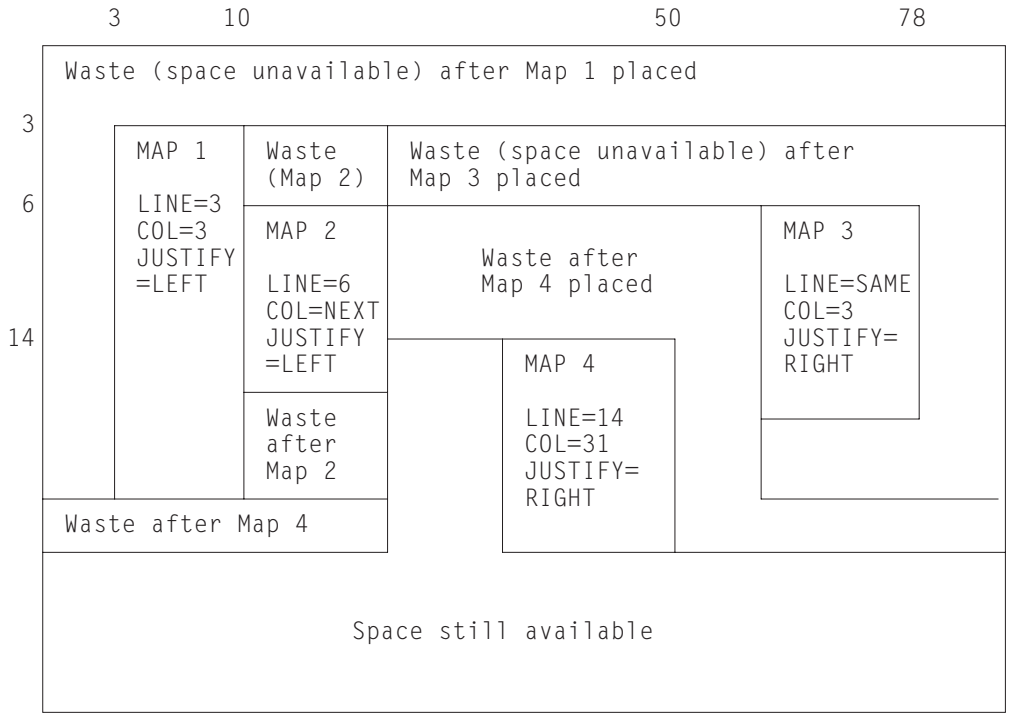


Figure 154. Successive placement of maps on a page, showing space made unavailable by each

5. 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 740. 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 on page 741.

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.

The following options apply 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.

Related reference:

RECEIVE MAP in Reference -> Application development

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.

As online systems have evolved, the emphasis has shifted steadily to making things as easy 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.

Follow the suggestions to reduce path length and line time. 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. The path length for a given task 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, and path length for any given command can 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 in “Building the output screen” on page 716, 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 typically 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 “Sharing data across transactions” on page 261 for ways to save the restart data.
- Use commands that are on the BMS “fast path” if possible. (See Minimum BMS in “BMS support levels” on page 687 for the commands and terminal types that qualify.)
- Use terminal control commands for very simple inputs an outputs, where you do not need BMS formatting or other function. If path length is critical, you

might 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 typically 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 might 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 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).
- 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 the DATAONLY option in “Merging the symbolic and physical maps” on page 715). 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.)

Text output

If you are sending text output to a terminal, and you do not need to format the screen for subsequent input, you do not need to create a map. BMS provides the **SEND TEXT** command expressly for this purpose.

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 want. Page size is determined as it is for other BMS output (see “The SEND PAGE command” on page 736).

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."

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 "Building logical messages" on page 735). 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 "BMS logical messages" on page 734).

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 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 720.

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 want. (Header format and page numbering are explained in "Header and trailer format" on page 748.)
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," 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 you 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 748).
7. BMS then goes to a new page and repeats from step 2 with the remaining data.

Text lines

BMS uses well-defined rules to break text into lines for display on a terminal.

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 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 664.

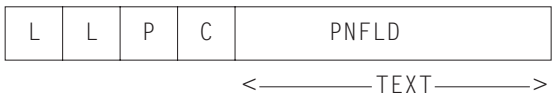
2. BMS copies your text character for character, including all blanks, with two exceptions that occur at line end:
   - If a line ends in the middle of a word, BMS fills out the current line with blanks and places the word that would not fit in the first available position of the next line. For this 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 660 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 559.) 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 or trailer on the pages of a text message, you point to a block of data with a set 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, 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 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 might have used either SEND MAP or SEND TEXT commands to build the page originally.

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 MAP commands, as long as each BMS SEND represents a separate page (that is, the ACCUM option is not used). The

usual restriction against mixing text with mapped output in the same logical method does not apply here, because the page is already formed.

You can also use these commands in a routing environment. 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. 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).

Related information:

"Using SET" on page 720
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.

"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.

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 738 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 667). It is the basis for the CICS-supplied transaction CMSG, with which terminal users can send messages to other terminals and users. See CMSG -message switching in 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 758.

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 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.

Message destinations

Destinations can be specified for your routed message in several ways: by requesting certain classes of operator, naming a particular operator or naming a particular terminal.

- 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 three-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 *can* 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 get your message. A terminal must be on the list to receive the message, but its presence there does not guarantee delivery. There might be operator restrictions associated with the terminal and, because delivery occurs later in time, the status or even the nature of the terminal can 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 might not have all the terminal definitions you expect. First, terminals that are autoinstalled might 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 might not be known to another. It depends on how they are defined, as explained in TYPETERM definition attributes in the *CICS Resource Definition Guide*. 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 typically 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 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 do not specify the operator class or a route list, BMS puts every terminal that meets the first two conditions 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 ROUTEDMSGS option in the terminal definition. Three values are possible: a terminal can 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 might 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 enters 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 previously described.

  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.

Table 53. 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 in “Logical device components” on page 770 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 54. 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 want. 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 in "Logical device components" on page 770).

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

A message can be delivered some time after the **ROUTE** command is issued, depending on the scheduling options in the 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 processed. 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. 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.
- 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 751 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.

Delivery is retried 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.

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.

Related information:

"Logical message recovery" on page 739
Logical messages created with a disposition of PAGING are kept in CICS temporary storage between creation and delivery.

Message identification

You can assign a title to your routed message if you want. 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 might 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 CSPG - page retrieval in the *CICS Supplied Transactions*.)

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 can 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 155. Assigning a title

Programming considerations with routing

For the most part, you build a routed message in the same way you do a non-routed message. However, because BMS builds a separate logical message for each terminal type among your destinations, there are differences.

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 740). 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.

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.

Without routing, you cannot use BMS SENDs, as noted in "Building logical messages" on page 735.

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 and the SEND MAP command, 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 the same way as for minimum BMS, and there is no need to change mapset definitions or to regenerate mapsets.

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 typically 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, 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 releases it by using a FREEMAIN request. It is advisable to release these storage areas, by using a FREEMAIN request, 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 releases it by using a FREEMAIN request. It is advisable to release these storage areas, by using a FREEMAIN request, for long-running transactions, but CICS frees these areas when the task terminates.

Related reference:

RECEIVE MAP in Reference -> Application development

SEND MAP in Reference -> Application development

Sample assembler MAPPINGDEV application

This example uses a modified version of the FILEA operator instruction sample to show you how to code the keywords associated with the MAPPINGDEV facility.

Figure 156 on page 762 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 use 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$$AGA') FROM(DFH$AGAO) ERASE
               MAPPINGDEV(EIBTRMID) SET(R6)
         MVC   OUTAREA(256),0(R6)
         MVC   OUTAREA+256(256),256(R6)
         EXEC  CICS SEND TEXT MAPPED FROM(OUTDATA) LENGTH(OUTLEN)
*
         EXEC CICS RECEIVE INTO(INAREA) LENGTH(INLEN)
              MAXLENGTH(MAXLEN)
*
         EXEC CICS RECEIVE MAP('DFH$AGA') SET(R7) LENGTH(INLEN)
              MAPPINGDEV(EIBTRMID) FROM(INAREA)
              CURSOR(820) AID(=C'3')
*
         XC    PROOF,PROOF
         MVC   PROOF(25),=C'You just keyed in number '
         MVC   PROOF+25(6),KEYI-DFH$$AGAI(R7)
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 156. ASM example of a MAPPINGDEV application

Partition support

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*.

Standard BMS is required for partitions.

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 manuals (*IBM 3290 Information Display Panel Description and Reference* for the 3290 and *IBM 8775 Display Terminal Component Description*) 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 767.
- 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 765.
- 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 766.

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 769).

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.

Uses for partitioned screens

Partitioned screens are particularly useful in certain types of application.

Scrolling

For transactions that produce more output than fits on a single screen, scrolling is an alternative to BMS terminal paging.

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 is 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 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

You can enhance usability if you partition a screen and allocate one area to error messages and other explanatory text. If you do this, 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 a 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 766)
- The partition set suffix, used to associate the partition set with a particular screen size (see "Establishing partitioning" on page 766)

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*.

Because these are assembler macros, you need to follow assembler format rules in creating them. See "Writing BMS macros" on page 695 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.

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 CHARSZE, 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 CHARSZE dimension; the partition width is the product of the number of columns and the horizontal CHARSZE value.

If you do not specify a CHARSZE size in your DFHPDI partition definition, BMS uses the default given in the DFHPSD partition set definition. If DFHPSD does not specify CHARSZE 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 705).

Partition options for BMS SEND commands

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 705. 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 734.

ASSIGN options for partitions

In addition to the INPARTN option, 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 757). 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 770). In addition, you cannot use partitions in combination with GDDM, although you can use partitions with outboard formats (see "Outboard formatting" on page 775).

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 . 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 728).

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 theBMS macros in Reference -> Application development manual with the commands to which they apply.

Terminal sharing

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, but you can interleave the component tasks of several pseudoconversational transaction sequences at a partitioned terminal.

To take a 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 764). You could modify the application to work on two records at once, or you could 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 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 the destroy the partition set required by another. Also, if two different CICS systems can 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 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 225.

Support for special hardware

In addition to partitions, BMS supports other special hardware features.

BMS supports the following 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**.

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 the *CICS Resource Definition Guide* for descriptions of both 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 737.

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 768). 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 CSPG - page retrieval in the *CICS Supplied Transactions* 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.

The following features are supported:

- 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 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 might want 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 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 want.

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, or ampersand.

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 in "Building the output screen" on page 716). 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 pressing 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 can 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 function key), the I subfield contains the contents of the field if the MDT is on and the L subfield reflects its length, as typical, 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 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 function 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. 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 Chapter 30, “Using batch data interchange,” on page 545 for more information about programming some of the devices that support outboard formatting.

BMS: design for performance

When building a formatted data stream with basic mapping support (BMS), you should bear in mind the factors described here.

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 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 710 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 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.

For more information, see “Efficient data set operations” on page 517. You might 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 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, many 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 might 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 can 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 typical 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 might 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 previous 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 START in Reference -> Application development 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.

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 (two-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” on page 660), although of a different type from the BMS copy facility. It also has destination restrictions that do not apply to BMS copying.

Part 16. Appendixes

© Copyright IBM Corp. 1989, 2013

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:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, 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 want 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.

© Copyright IBM Corp. 1989, 2013

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. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Bibliography

CICS books for CICS Transaction Server for z/OS

General

CICS Transaction Server for z/OS Program Directory, GI13-0565
CICS Transaction Server for z/OS What's New, GC34-7192
CICS Transaction Server for z/OS Upgrading from CICS TS Version 3.1, GC34-7188
CICS Transaction Server for z/OS Upgrading from CICS TS Version 3.2, GC34-7189
CICS Transaction Server for z/OS Upgrading from CICS TS Version 4.1, GC34-7190
CICS Transaction Server for z/OS Installation Guide, GC34-7171

Access to CICS

CICS Internet Guide, SC34-7173
CICS Web Services Guide, SC34-7191

Administration

CICS System Definition Guide, SC34-7185
CICS Customization Guide, SC34-7161
CICS Resource Definition Guide, SC34-7181
CICS Operations and Utilities Guide, SC34-7213
CICS RACF Security Guide, SC34-7179
CICS Supplied Transactions, SC34-7184

Programming

CICS Application Programming Guide, SC34-7158
CICS Application Programming Reference, SC34-7159
CICS System Programming Reference, SC34-7186
CICS Front End Programming Interface User's Guide, SC34-7169
CICS C++ OO Class Libraries, SC34-7162
CICS Distributed Transaction Programming Guide, SC34-7167
CICS Business Transaction Services, SC34-7160
Java Applications in CICS, SC34-7174

Diagnosis

CICS Problem Determination Guide, GC34-7178
CICS Performance Guide, SC34-7177
CICS Messages and Codes Vol 1, GC34-7175
CICS Messages and Codes Vol 2, GC34-7176
CICS Diagnosis Reference, GC34-7166
CICS Recovery and Restart Guide, SC34-7180
CICS Data Areas, GC34-7163
CICS Trace Entries, SC34-7187
CICS Debugging Tools Interfaces Reference, GC34-7165

Communication

CICS Intercommunication Guide, SC34-7172
CICS External Interfaces Guide, SC34-7168

Databases

CICS DB2 Guide, SC34-7164
CICS IMS Database Control Guide, SC34-7170

© Copyright IBM Corp. 1989, 2013

CICS Shared Data Tables Guide, SC34-7182

CICSPlex SM books for CICS Transaction Server for z/OS

General

CICSPlex SM Concepts and Planning, SC34-7196
CICSPlex SM Web User Interface Guide, SC34-7214

Administration and Management

CICSPlex SM Administration, SC34-7193
CICSPlex SM Operations Views Reference, SC34-7202
CICSPlex SM Monitor Views Reference, SC34-7200
CICSPlex SM Managing Workloads, SC34-7199
CICSPlex SM Managing Resource Usage, SC34-7198
CICSPlex SM Managing Business Applications, SC34-7197

Programming

CICSPlex SM Application Programming Guide, SC34-7194
CICSPlex SM Application Programming Reference, SC34-7195

Diagnosis

CICSPlex SM Resource Tables Reference Vol 1, SC34-7204
CICSPlex SM Resource Tables Reference Vol 2, SC34-7205
CICSPlex SM Messages and Codes, GC34-7201
CICSPlex SM Problem Determination, GC34-7203

Other CICS publications

The following publications contain further information about CICS, but are not provided as part of CICS Transaction Server for z/OS, Version 5 Release 1.

Designing and Programming CICS Applications, SR23-9692

CICS Application Migration Aid Guide, SC33-0768

CICS Family: API Structure, SC33-1007

CICS Family: Client/Server Programming, SC33-1435

CICS Family: Interproduct Communication, SC34-6853

CICS Family: Communicating from CICS on System/390, SC34-6854

CICS Transaction Gateway for z/OS Administration, SC34-5528

CICS Family: General Information, GC33-0155

CICS 4.1 Sample Applications Guide, SC33-1173

CICS/ESA 3.3 XRF Guide , SC33-0661

Other IBM publications

The following publications contain information about related IBM products.

MVS

z/OS MVS Initialization and Tuning Guide, SA22-7591
z/OS MVS Initialization and Tuning Reference, SA22-7592
z/OS MVS JCL Reference, SA22-7597
OS/390 V2R8.0 MVS System Commands, GC28-1781

SNA

Sessions between Logical Units, GC20-1868.

SQL

DB2 Universal Database for OS/390 and z/OS: Application Programming and SQL Guide, SC26-9933
DB2 Universal Database for OS/390 and z/OS: SQL Reference, SC26-9944.

Other related books

An Introduction to the IBM 3270 Information Display System, GA27-2739
3274 Control Unit Reference Summary, GX20-1878
Component Description: IBM 2721 Portable Audio Terminal, GA27-3029
IBM 2780 Data Transmission Terminal Component Description, GA27-3035
CICS/ESE 3.3 IBM 3270 Data Stream Device Guide, SC33-0232
IBM 3270 Data Stream Programmer's Reference, GA23-0059
IBM 4700/3600/3630 Guide, SC33-0233

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.

© Copyright IBM Corp. 1989, 2013

Index

Special characters

#echo command 619, 620
#include command 620
#set command 606, 619, 620
REQID option 506

Numerics

10/63 magnetic slot reader 772
31-bit addressing
  assembler language 61
  COBOL 24
31-bit mode transaction 306
3262 printer 659
3270 bridge
  ADS descriptor 699
3270 display 527
3270 family
  AID 562
  attention identifier 562
  attention keys 562
  attributes, extended 556
  base color 554
  buffer 551
  color, base 554
  data stream 551
  data stream orders 556
  data stream, outbound 559
  display characteristics 553
  extended attributes 556
  field attributes 554
  field format, inbound 564
  fields 553
  inbound field format 564
  input from 562
  intensity 554
  MDT 554
  modified data tag 554
  orders in data stream 556
  outbound data stream 559
  protection 554
  reading from 563
  terminal, writing to 552
  unformatted mode 565
  write control character 552
  writing to terminal 552
3270 Information Display System 152
3270 printer 660
  options 661
3270 screen field 727
3289 printer 659
3290 display 763
3601 logical unit 770
3770 batch data interchange logical unit 770
3770 batch logical unit 770
3790 batch data interchange logical unit 770
64-bit addressing mode
  assembly language 75

A

abend 354
ABEND command 361
abend exit facility 361
abend exit program 363
abend exit routine 363
abend user task, EDF 182
abnormal termination recovery 361
ACCEPT statement, COBOL 24
access to system information
  EXEC interface block (EIB) 5
ACCUM option 662, 735
ACK 525
acknowledgment 525
active partition 767
ACTPARTN option 735, 767
adding records 514
ADDRESS command 5
ADDRESS COMMAREA command 303
ADDRESS special register 29
addressing mode (AMODE)
  options for CICS applications 126
addressing of CICS areas 777
ADS descriptor 699
affinity 312
  recommended programming techniques 314
  safe programming techniques 314
  suspect programming techniques 314
  unsafe programming techniques 314
AFFINITY attribute 339
affinity groups 338
affinity relations
  BAPPL 339
  LOCKED 342
  LUname 343
  userid 344
AFFLIFE attribute 339
AFTER option 570
agile service delivery 105, 107
ALARM option 714
ALLOCATE command 259
  inhibit wait, NOSUSPEND option 259
ALLOCERR condition 676
alternate
  index 481
  key 481
ALTPAGE value 739
AMODE (addressing mode)
  options for CICS applications 126
AMODE(64)
  assembly language 75
APAK transaction 665
APCG 253
API
  subset for DPL 379
APOST option 85
APPLCTN 106
APPLDEF 106
application
  getting started 107
  transform to XML 389
application bundle 106
application entry point
  getting started 107
application program logical levels 31
application programs
  asynchronous processing 383
  design 223
  design for performance 252
  distributed program link 371
  distributed transaction processing 383
  function shipping 371
  installing 117
  intercommunication considerations 369
  logical levels 301
  testing 161
  transaction routing 370
  using BMS map sets 132
  writing 4
application project 106
application/x-www-form-urlencoded 606
applications
  deploying 106
  enabling 106
  installing 106
area, dynamic storage 253
argc 46
argv 46
ASIS option 728
ASKTIME command 569
assembler language 61
  31-bit addressing 61
  64–bit registers 61
  applications 255
  CALL statement 71
  Language Environment requirements 69
  mixing with other languages 15, 71
  non-conforming routines 69
  programming techniques 61, 69, 71
  restrictions 61
  working storage 61
Assembler language 99
  DFHECALL macro 99
assembly 79
assembly language 61
  64-bit addressing mode applications 75
  programming techniques 71
  working storage 61
assembly, TYPE=DSECT 698
ASSIGN command 5, 535, 666
  DESTCOUNT option 757
  MAPCOLUMN option 743
  MAPHEIGHT option 743
  MAPLINE option 743

© Copyright IBM Corp. 1989, 2013

ASSIGN command *(continued)*
MAPWIDTH option 743
MSR option 772
options 535
PAGENUM option 758
assistant, XML 389
asynchronous journal output 346
asynchronous processing 369, 383
AT option 570
ATI 225, 524, 599
ATNI 371
Atom feed
XML binding
creating 455
attention field 775
attention identifier 728
ATTENTION key 525
automatic task initiation 225, 524
automatic transaction initiation 599
AUTOPAGE option 737
auxiliary temporary storage 245
auxiliary trace 258
AZI6 371

B

backout of resources 348
BAKR (branch and stack) assembler instruction 61
BASE option 710
basic mapping support
assembling and link-editing physical map sets 152
assembling map 697
assembly, TYPE=DSECT 698
basic mapping support
standard 687
BMS support levels 687
complex fields 700
composite fields 700
copy facility 779
creating map 691
cursor position 719
cursor, finding the 729
data streams 776
data, moving to map 711
DFHASMVS procedure 152
DFHLNKVS procedure 152
DFHMAPS, procedure for installing maps 157
DFHMDF macro 691, 692
DFHMDI macro 691, 694
DFHMSD macro 691, 694
DFHPSD, for defining partition sets 158
display, receiving data from 722
EOC condition 733
field, group 700
fields 690
fields, complex 700
fields, composite 700
fields, repeated 701
finding the cursor 729
full 687
group field 700
GRPNAME option 700
initializing output map 710
basic mapping support *(continued)*
installing mapsets 149, 150
installing partition sets 158
installing physical map sets 152
installing symbolic description map sets 154
invalid data 720
link-edit 698
macro 691
macros, rules for writing 695
map 689
map sets 699
map, assembling 697
map, creating 691
map, initializing output 710
map, moving data to 711
map, physical 698
map, symbolic 698
mapping input data 725
maps 776, 777, 779
maps, storage 709
MAPSET resource definition 698
MDT 727
message lengths, reducing 745
minimizing path length 744
minimum 687
modified data tag (MDT) 776
moving data to map 711
multimap screens 779
OCCURS option 701
output example 689
output map, initializing 710
page building operations 779
page routing operations 779
path length, minimizing 744
performance considerations 744
physical map 698
physical maps 151
preparing maps 149
PROGRAM resource definition 698
receiving data from display 722
reducing message lengths 745
repeated fields 701
rules for writing macros 695
screen copy 673
SEND MAP command 708
storage for maps 709
support across platforms 687
symbolic description map sets for BMS 154
symbolic map 698
symbolic maps 151
terminals supported 687
TYPE=DSECT assembly 698
types of mapsets 151
uppercase translation 728
using BMS map sets in application programs 132
using symbolic map sets in a program 154
batch compilation for COBOL programs 39
batch data interchange 545
definite response 546
DEFRESP option 546
destination identification 546
ISSUE WAIT command 546
batch data interchange *(continued)*
NOWAIT option 546
batch utility
XML assistant 389
BDAM 517
browsing operations 509
data sets 492, 493
exclusive control 512
updating operations 512
BDI 659
BGAM 521
big COMMAREAs 267, 270, 288, 295
binding file
XML binding 455
bindings 473, 475
blank fields 777
block references 493
blocked-data set 492
BMS 659
routing 670
BMS commands 519
bookmarks 629, 630
BOTTOM command, CEBR transaction 194
BRACKET option 541
bracket protocol, LAST option 541
bridge (3270)
ADS descriptor 699
brightness 774
browse operation
BDAM 509
BROWSE TEMP STORAGE option, CEDF 183
browsing 518
DELAY 518
records 506
SUSPEND 518
BTS activities 286
BUILDCHAIN 539
business services 469

C

C and C++ 45
arguments 50
EIB, accessing 51
locale support 52
mapping to XML Schema 420, 422
mixing with other languages 15
programming techniques 46
restrictions 46
support 45
working storage 45
XPLink 52, 53, 54
C language considerations
LENGTH option default 249
struct, symbolic description map set 151
caching of document templates 609
CALL DL/I interface, COBOL 27
CALL statement
assembler language 71
callable services 12
CANCEL command 569
CBLCARD option 85
CBLPSHPOP runtime option for Language Environment 27

CDUMP 46
CEBR transaction 191
  BOTTOM command 194
  browse transaction 191
  CEBR initiation 191
  COLUMN command 195
  displays 193
  FIND command 195
  GET command 195
  initiation 191
  LINE command 195
  PURGE command 196
  PUT command 196
  QUEUE command 196
  security considerations 191
  SYSID command 196
  temporary storage browse 191
  TERMINAL command 196
  TOP command 196
  transient data 197
CECI transaction
  about to execute command 201
  ampersand (&) 206
  body 203
  command execution complete 202
  command input 199
  command input line 199
  command line 199
  command syntax check 201
  ENQ commands 209
  function key values area 204
  information area 203
  introduction 199
  making changes 205
  message line 204
  program control 209
  screen layout 199
  shared storage 209
  status area 200
  terminal sharing 209
  variables 206
CECS transaction 205
CEDF transaction 165, 166
  abend user task 183
  body 169
  browse temporary storage 183
  display register 184
  displays 168
  DPL 179
  dual-screen mode 178
  EDF transaction 166
  function key 168
  functions 165
  header 169
  invoke CECI 183
  invoking 166
  modifying execution 180
  non-terminal transactions 179
  options on function (PF) keys 182
  overtyping displays 180
  program labels 180
  pseudoconversational programs 178
  remote transactions 178
  remote-linked programs 179
  security 166
  single-screen mode 176
CEEBINT, Language Environment HLL user exit 22
CEEBXITA user exit 20
CEECSTX user exit 20
CEEDOPT CSECT 17
CEEENTRY macro 69
CEEHDLR service 13
CEEROPT CSECT 17
CEEUOPT CSECT 17
CEEWUCHA sample user condition handler 28, 57
CESF, GOODNIGHT transaction 544
chaining 527
chaining of data 539
CHANNEL
  LINK command 300
  option 266, 300, 304
  RETURN command 300
  XCTL command 300
channel-based services 473, 476
channels
  as large COMMAREAs 267
  basic examples 268
  benefits of 295
  compared to BTS activities 286
  constructing 285
  creating 273
  current 274, 278
  current, example, with LINK commands 274
  current, example, with XCTL commands 277
  data conversion 291
  designing 284
  discovering which containers were passed to a program 283
  discovering which containers were returned from a link 283
  dynamic and distributed routing 288
  LINK command 307
  migrating 296
    dynamically-routed applications 299
    LINK command 297
    RETURN command 298
    temporary storage 299
    XCTL command 297
  on LINK and XCTL commands 245
  on RETURN commands 266
  overview 267
  read only containers 283
  RETURN command 309
  scope of 279, 281
  typical scenarios
    multiple interactive components 272
    one channel—one program 270
    one channel—several programs 271
    several channels, one component 271
  using from JCICS 288
channels as large COMMAREAs 267, 270, 288, 295
character set 610
checkout, program 165
child enclaves 17
CICS
  testing environment 6
CICS areas, addressing 777
CICS dump utility program 367
CICS file
  as document template 614
CICS option 85
CICS printer 659
  determining characteristics of 666
CICS program
  as document template 615
    DFHDHTL macro 615, 616
CICS-key storage 581
CICS-maintained table 495
CICS-value data areas 249
CICSDATAKEY option 244, 584
CICSVAR environment variable 20
CLASS option 679
CLEAR
  key 769
  PARTITION AID value 769
  PARTITION key 769
CLEAR key 266
client code page 610
client region 372
CLOCK 46
cloud 105
cloud applications 105
cloud enablement 107
CMT 495
CNOTCOMPL option 539
COBOL 23
  31-bit addressing 24
  ADDRESS special register 29
  addressing CICS data areas 29
  batch compilation 39
  blank lines 37
  CALL DL/I interface 27
  calling subprograms 30, 33
  CBLPSHPOP runtime option 27
  compiler options 24
  example of DFHNCTR call 650
  global variables 37
  mapping to XML schema 415
  mapping to XML Schema 411
  mixing with other languages 15
  nested programs 41
  programming restrictions VS COBOL II 28
  programming techniques 24, 29, 33
  reference modification 37
  REPLACE statement 37
  reserved word table 24
  restrictions 24, 33, 37, 255
  run unit 31
  support 23
  translation 37
  variable repeating content 450
  WITH DEBUGGING MODE 24
  working storage 23
COBOL2 option 85
COBOL3 option 85
code pages for CICS documents 610
CODEREG operand 65
COLUMN
  command, CEBR transaction 195
COM assembler instruction 61

command language translator 81, 94
APOST option 85
CBLCARD option 85
CICS option 85
COBOL2 option 85
COBOL3 option 85
CPSM option 85
DBCS option 85
DEBUG option 85
DLI option 85
EDF option 85
EPILOG option 85
EXCI option 85
FLAG option 85
GDS option 85
GRAPHIC option 85
LEASM option 85
LENGTH option 85
line numbers 83
LINECOUNT option 85
LINKAGE option 85
MARGINS option 85
NATLANG option 85
NOCBLCARD option 85
NOCPSM option 85
NODEBUG option 85
NOEDF option 85
NOEPILOG option 85
NOFEPI option 85
NOLENGTH option 85
NOLINKAGE option 85
NONUM option 85
NOOPSEQUENCE option 85
NOOPTIONS option 85
NOPROLOG option 85
NOSEQ option 85
NOSEQUENCE option 85
NOSOURCE option 83, 85
NOSPIE option 85
NOVBREF option 85
NOXREF option 85
NUM option 85
OPMARGINS option 85
OPSEQUENCE option 85
options 94, 96
OPTIONS option 85
PROLOG option 85
QUOTE option 85
SEQ option 85
SEQUENCE option 85
SOURCE option 83, 85
SP option 85
SPACE option 85
SPIE option 85
SYSEIB option 85
VBREF option 83, 85
XOPTS keyword 94
XREF option 85
command-level interpreter
ENTER key 204
invoking 205
security considerations 199
command, SYNCPOINT 348
COMMAREA 224, 243, 244, 254
LINK command 300
migrating to channels and containers 296
COMMAREA *(continued)*
dynamically-routed applications 299
LINK command 297
RETURN command 298
XCTL command 297
option 265, 300, 302, 304
COMMAREAs > 32K 267, 288, 295
common work area (CWA) 261
protecting 262
communication area for a document template 617
Communications Server 523
compilation 79
compilers supported
Language Environment 11
complex fields 700
component architecture 469
components
multiple, interactive 272
one channel—several programs 271
several channels, one component 271
composite 470
composite fields 700
condition, exception 350
CONNECT PROCESS command 381
CONSISTENT option
READ command 503
constructing a channel 285
containers
basic examples 268
context, BTS or channel 287
designing a channel 284
discovering which containers were passed to a program 283
discovering which were returned from a link 283
migrating 296
dynamically-routed applications 299
LINK command 297
RETURN command 298
temporary storage 299
XCTL command 297
overview 267
read only 283
using from JCICS 288
content type mapping xi
content types xi
contention for resources 224
contention for terminal 524
context
of containers, BTS or channel 287
control
exclusive of BDAM 512
of VSAM blocks 488
conversation partner 523
conversational programming 223, 549
CONVERSE command 527, 543, 549
copy facility
BMS 779
COPY statements 98
copybook translation 98
counter name
named counters 638
coupling facility data tables 495
coupling facility list structure
current value 638
CPI
references 4
CPI Communications interface module, DFHCPLC 98
CPI Communications stub 384
CPI-C 369, 384
CPSM option 85
CQRY transaction 226
creating a channel 273
CSECT, adding to map assembly 157
CSNAP 46
CSPG transaction 670, 673, 736, 738
CSPP transaction 226
CTDLI 46
CTEST 46
CTLCHAR option 662
CTRACE 46
current channel
example, with LINK commands 274
example, with XCTL commands 277
overview 274, 278
CURSOR option 719, 735
ACCUM option 713
SEND MAP command
ACCUM option 713
cursor position 719
cursor positioning, symbolic 719
cursor-detectable field 774
cursor, finding the 729
CVDA 249, 368, 369
CWA 261
CWAKEY parameter 262

D

data
chaining 539
definition 254
initialization 254
passing to other program 302
records 249
storing in transaction 243
data conversion 289
and channels 289
a SOAP example 294
why necessary 289
with channels 291
data interchange block 83
data sets 517
access from CICS application programs 502
batch data interchange 545
BDAM 492, 493
blocked 492
empty 481
sequential 260
user 249
data storing in transaction 243
data streams
compressing 778
inbound 777
RA order 778
repeat-to-address orders (SBA) 778
SBA order 778
set buffer address order 778

data tables
coupling facility 495
shared 495
data, moving to map 711
data, reading from a display 725
DATAONLY option 713, 715, 778
DATAREG operand 65
date field of EIB 5
DBCS option 85
DDname list, in translator dynamic invocation 93
DDS 706
deadlock 257
prevention 488
deadlocks 502
DEBKEY option 509
deblocking argument 493
DEBREC option 493, 509
DEBUG option 85
debugging 165
default
action for condition 350
deferred journal output 347
definite response protocol
terminal control 540
DEFRESP option 546, 549
terminal control 540
DELAY command 569, 570
DELETE statement, COBOL 24
DELETEQ TD command 597
DELETEQ TS command 601
deleting records 512
DEQ command 573
DEQUEUE command 669
design considerations of applications
exclusive control of resources 257
designator character 774
designing a channel 284
DESTCOUNT option 757
DESTID option 546
DESTIDLENG option 546
destination identification 546
detectable field 774
device dependent support 706
device-dependent maps 705
DFH3QSS 6
DFHAID 46
DFHAPXPO 17
DFHASMVS procedure 152, 154, 158
DFHBMSCA 46, 713
DFHBMSCA definitions 729
DFHBMSUP 700
DFHCOMMAREA 24, 302
DFHCPLC 384
DFHCPLC, CPI Communications interface module 98
DFHCPLRR, SAA Resource Recovery interface module 98
DFHDHTL macro 615, 616
DFHDHTXD 617
DFHDHTXH 617
DFHDHTXL 617
DFHDHTXO 617
DFHDYPDS 296
DFHEAG, interface module for assembler 104
DFHEAG0, interface module for assembler 104
DFHEAI, interface module for assembler 99
DFHEAI0, interface module for assembler 99
DFHEAP1$, translator for assembler 84
DFHECP1$, translator for COBOL 84
DFHEDF group 162
DFHEDP1$, translator for C 84
DFHEGTAL procedure 133, 135
DFHEIBLK 24
DFHEIEND macro 90, 92
DFHEIENT macro 71, 90, 92
CODEREG 65
DATAREG 65
defaults 65
EIBREG 65
DFHEIPLR symbolic register 65
DFHEIRET macro 71, 86, 90
DFHEISTG macro 90, 92
DFHEITAL procedure 133, 135
DFHEIVAR 24
DFHELII 99
DFHELII, interface module for Language Environment conforming compilers 99
DFHEPP1$, translator for PL/I 84
DFHEXEC 99
DFHEXTAL procedure 133, 135
DFHFCT macro 495
DFHLNKVS procedure 152, 158
DFHLS2SC
cataloged procedure 390
DFHLS2SC, CICS XML assistant program 455
DFHMAPS procedure 132, 155
DFHMAPT
procedure for installing HTML templates 156
DFHMDF macro 691, 692
display characteristics 711
DSATTS option 711
MAPATTS option 711
DFHMDI macro 691, 694
DFHMIRS program 374
DFHMSCAN utility program 118
DFHMSD macro 691, 694
BASE option 710
STORAGE option 710
DFHMSD, macro for assembling map sets 151
DFHMSRCA 46, 772
DFHNC001
default named counter pool 638
DFHNCO macro
named counter options table 638
DFHNCOPT
named counter options table 638
DFHNCTR
example COBOL call with null pointers 650
DFHPDI macro 765
DFHPEP program 361
DFHPSD macro 158, 765
DFHRESP translator function 83, 351
DFHSC2LS
cataloged procedure 396
DFHURLDS 753
DFHVALUE 83
DFHYITDL procedure 133, 143
DFHYITEL procedure 133, 143
DFHYITPL procedure 133, 141
DFHYITVL procedure 133, 138
DFHYXTDL procedure 133, 143
DFHYXTEL procedure 133, 143
DFHYXTPL procedure 133, 141
DFHYXTVL procedure 133, 138
DFHZITCL procedure 133, 138
DFHZITPL procedure 133, 141
diagnosing SCA problems 478
DIB 83
direct terminal 758
discovering which containers were passed to a program 283
discovering which containers were returned from a link 283
display
register, EDF 184
screens 266
display characteristics 711
DISPLAY statement, COBOL 24
display, reading from 725
distributed application design 229
distributed program link 300
client region 372
COMMAREA option 374
DPL API subset 379
exception conditions 381
independent syncpoints 374
options 372
programming considerations 379
REMOTENAME option 374
REMOTESYSTEM option 374
server program 374
server region 372, 374
SYSID option 374
TRANSID option 374
DL/I
references 4
syncpoints 349
DLI option 85
DLLs 17
DOCTEMPLATE resource definition 605, 613
EXITPGM attribute 617
FILE attribute 614
HFSFILE attribute 614
MEMBERNAME attribute 613
PROGRAM attribute 615
TDQUEUE attribute 614
TSQUEUE attribute 614
DOCTOKEN 623, 634
DOCUMENT CREATE command 605, 623
DELIMITER option 624, 626
DOCSIZE option 632
DOCTOKEN 623
LISTLENGTH option 624
SYMBOLLIST option 605, 606, 623, 624, 626
DOCUMENT DELETE command 634
DOCTOKEN 634
DOCUMENT INSERT command 605, 629, 630, 632

DOCUMENT RETRIEVE command 605, 632
CLNTCODEPAGE option 610, 632
HOSTCODEPAGE option 610
DOCUMENT SET command 605
DELIMITER option 624, 626
LISTLENGTH option 624
SYMBOL option 605, 606, 624, 626
SYMBOLLIST option 605, 606, 624, 626
document templates 605
and CICS Web support 605, 610, 614
caching 609
specifying in exit programs 617
CICS file 614
CICS program 615, 616
communication area 617
DFHDHTL macro 615, 616
embedded commands
#echo 619, 620
#include 620
#set 606, 619, 620
exit program 617
HFS file 614
partitioned data set (PDS) 613
placing in documents 623, 629, 630
setting up 613
symbol lists 606
symbols 605, 606, 619
temporary storage queue 614
transient data queue 614
documents 605
adding data 629
and CICS Web support 605
bookmarks 629, 630
code page conversion 610
creating 623
deleting 634
replacing data 630
retrieving 632
reusing 632
DPL 179, 300, 349, 369, 371
DSA 253
DSATTS option 711
DTP 369, 383
DUMP TRANSACTION command 367
DUPKEY condition 506
dynamic
program 256
storage area 253
transaction backout program 361
transaction routing 304
dynamic invocation of translator 84
Dynamic LIBRARY 118
Dynamic Link Libraries 17
dynamic program link
affinity 342
dynamic routing with channels 288
dynamic storage, extensions 73
dynamically-routed applications
migrating to channels and containers 299

E

ECBLIST 573
EDF 83, 86, 165
EDF option 85
EIB 83, 350, 524
description 5
EIBCALEN field 303
EIBCOMPL field 527
EIBFN field 304
terminal control feedback 536
EIBREG operand 65
empty data sets 481
end-of-data indicator character 544
ENDBR command 506
ENQ command 259, 573
ENQBUSY condition 259
ENQUEUE command 669
enqueuing
in a VSAM file 257
VSAM internal 257
entry point, trace 366
entry-sequenced data set (ESDS) 481
EOC condition 539, 733
EODI character 544
EODS condition 539
EPILOG option 85
EQUAL option 503
ERASE option 662, 714, 735
ERASEAUP option 714, 735, 746
ERDSA 129
escape sequence 606, 626
ESDS
upgrading to extended addressing 486
ESDS (entry-sequenced data set) 481
events
monitoring point 367
examples
channels, basic 268
CICS client program that contructs a channel 286
CICS server program that uses a channel 286
containers, basic 268
multiple interactive components 272
one channel—one program 270
one channel—several programs 271
several channels, one component 271
simple client program compared to a BTS activity 286
exception condition
description 350
exception conditions
HANDLE CONDITION command 354, 356
IGNORE CONDITION command 358
exception trace entry point 366
EXCI 87
CALL 384
communications 384
option 85
EXCI - external call interface 369
exclusive control of records
BDAM 512
VSAM 488
VSAM RLS 489
exclusive resources 257
EXEC interface block 83
EXEC interface modules 98, 99, 104, 145
EXEC interface stubs 99, 104
EXECKEY 244, 262
EXECKEY parameter 581
execution diagnostic facility 83, 86, 165
exit program
as document template 617
communication area 617
expiration time
specifying 570
extended read-only DSA (ERDSA) 129
extended relative byte address (XRBA) 485
extended relative byte addressing (XRBA)
upgrading 486
external call interface (EXCI) 369
External CICS interface (EXCI) 384
extrapartition queues 598
extrapartition transient data 248, 260
EYUVALUE 83

F

FEPI
references 4
FEPI - Front End Programming Interface 369
FETCH 46
FETCHABLE option 58
field
blank 777
group 700
fields
BMS 690
complex 700
composite 700
repeated 701
file control
BDAM data sets 493
overview 481
FIND command, CEBR transaction 195
finding the cursor 729
flag byte, route list 753
FLAG option 85
flip-flop mode 523
FLOAT compiler option 55
floating maps 739
FMH 541
inbound 541
option 541
outbound 541
FMHPARM option 735
FOR option 570
formatted screen, reading from a 727
FORMATTIME command 569
FORMFEED option 664, 735
FREE command 542
FREEKB option 714
FREEMAIN command 579
FROM option 713
Front End Programming Interface (FEPI) 369
FRSET option 714
function (PF) keys, CEBR transaction 194
function key 168, 204
function management header
description 541

function shipping 371
Function shipping 369
function-ship 511
functions, EDF 165

G

GDDM 716
GDS option 85
generic delete 512
generic key 503
GENERIC option 485, 503
GET command, CEBR transaction 195
GETMAIN command 244
  CICSDATAKEY option 244, 584
  INITIMG option 255, 579
  NOSUSPEND option 580
  SHARED option 243, 249, 579
  TASKDATAKEY option 244
  USERDATAKEY option 244, 584
GETMAIN64 command
  SHARED option 249
global user exits 584
global variables in COBOL 37
GOODNIGHT transaction, CESF 544
GRAPHIC option 85
group field 700
GRPNAME option 700
GTEQ option 485, 503

H

half-duplex mode 523
HANDLE ABEND 360
HANDLE ABEND command 354, 361
HANDLE ABEND LABEL restriction
  with assembler language 61
HANDLE AID command 728
HANDLE CONDITION command 354,
  360
HANDLE CONDITION ERROR
  command 358
HFS file
  as document template 614
HOLD option 300
HONEOM option 662
horizontal tabs 665
host code page 610
HTML templates
  installing 156

I

IBM Screen Definition Facility II (SDF
  II) 149
IBMWRLKC linkage editor input 57
ICTL (input format control) assembler
  instruction 61
ICVR parameter 162
identification
  BDAM record 493
  VSAM record 484
IGNORE CONDITION command 354,
  358
IGREQID condition 735
IGYCCICS 24
IGZWRLKA 28
IMMEDIATE option 304, 524, 541
IMS.RESLIB (IMS library) 145
INBFMH condition 539
inbound
  data streams 777
index, alternate 481
indirect queues 599
information center xi
information center content types xi
initializing output map 710
INITIMG option 255, 579
input data
  chaining of 539
input map, symbolic 724
INPUTMSG option 300, 304
INQUIRE command 5
INQUIRE TERMINAL command 536,
  666
INRTN option 768
installing application programs
  assembler language 135
  C 141
  COBOL 137
  PL/I 141
  using your own job stream 145
installing assembly application programs
  sample job stream for 135
Installing HTML templates 156
integrated CICS translator 79
integrated translators 6, 79
inter-transaction affinity
  affinity life times 338
  affinity transaction groups 338
  caused by application generators 337
  detecting 337
  programming techniques 314
  relations and lifetimes 339
    BAPPL relation 339
    global relation 340
    terminal relation 343
    userid relation 344
  safe programming techniques 316
    the COMMAREA 316
    the TCTUA 317
    using BTS containers 320
    using DEQ with
      ENQMODEL 319
    using ENQ with
      ENQMODEL 319
  suspect programming techniques
    DELAY and CANCEL REQID
      commands 334
    global user exits 313
    INQUIRE and SET
      commands 313
    POST command 336
    RETRIEVE WAIT and START
      commands 331
    START and CANCEL REQID
      commands 332
    transient data 330
  temporary storage 327
  temporary storage data-sharing 327
  unsafe programming techniques 321
    the CWA 321
    using DEQ 326
inter-transaction affinity (continued)
  unsafe programming techniques
    (continued)
    using ENQ 326
    using LOAD PROGRAM
      HOLD 322
    using shared storage 322
    using task lifetime storage 324
    using WAIT EVENT 325
interactive debugging
  CECI transaction 199
  CECS transaction 205
  CEDF transaction 165
interactive problem control system 367
intercommunication 369
interface modules
  CPI Communications 98
  EXEC 98
  programs using EXEC CICS or EXEC
    DLI commands 145
  SAA Resource Recovery 98
  using 135
interface stubs, EXEC 99, 104
interleaving conversation with message
  routing 758
interregion communication 384
interrupting 525
interval control 569
  cancel interval control command 569
  DELAY command 570
  delay processing of a task 569
  expiration time 570
  POST command 570
  specifying request identifier 569
  START command 570
  starting a task 569
INTERVAL option 570
intrapartition queues 597
intrapartition transient data 248
INVITE option 523
INVOKE SERVICE 473, 475, 476
invoking EDF 166
INVPARTN condition 769
INVPARTNSET condition 769
INVREQ condition 735
IPCS 367
IRC 384
iscics 6
ISCINVREQ 371
ISSUE ABORT command 545
ISSUE ADD command 545
ISSUE COPY command 525, 673
ISSUE DISCONNECT command 525
ISSUE END command 545
ISSUE ERASE command 525, 545
ISSUE NOTE command 546
ISSUE PRINT command 673
ISSUE QUERY command 545
ISSUE RECEIVE command 545
ISSUE REPLACE command 545
ISSUE SEND command 545
ISSUE WAIT command 546

J

JCICS
  and channels 288

JES 5, 674
JES (job entry subsystem)
  exits 678
  input 678
  RESP and RESP2 options 676
  retrieve data from JES spool 674
  spooler commands 676
Job Entry Subsystem component of MVS 674
journal
  records 249, 346
journal control
  output synchronization 346
journal identifier 348
journal type identifier 348
journaling 260, 346
JOURNALNAME 348
JOURNALNUM 348
JTYPEID 348
JUSFIRST option 747
JUSLAST option 747
JUSTIFY option 747

K

key
  alternate (secondary) 481
  generic 503
  hardware print 673
key-sequenced data set (KSDS) 481
keys
  physical 493
KSDS (key-sequenced data set) 481

L

L8 and L9 mode TCB 238, 239
  restrictions 241
Language Environment 11
  abend handling 13
    PL/I 57
  AID handling 13
  assembler language 69
  callable services 12
  CEEBINT 22
  CICSVAR environment variable 20
  compilers supported 11
  condition handlers for Language Environment 13
  condition handlers, user-written 13
  condition handling 13
  DLLs 17
  dump destination 12
  Dynamic Link Libraries 17
  HANDLE AID command 13
  HANDLE CONDITION command 13
  HLL user exit 22
  levels of support 11
  message destination 12
  mixing languages 15
  PL/I 57
  runtime options 17, 20
    and CICS LINK 17
    CBLPSHPOP 27
    CEEBXITA user exit 20
Language Environment *(continued)*
  runtime options *(continued)*
    CEECSTX user exit 20
    in child enclaves 17
  storage 14
  support, levels of 11
  VS COBOL II 28
language structure
  converting to WSDL 390
large COMMAREAs 267, 270, 288, 295
LAST option 542, 714
  bracket protocol 541
LDC 770
LDCMNEM option 770
LDCNUM option 770
LEASM option 85
LENGERR condition 527
LENGTH option 85, 249, 527
LENGTHLIST option
  multiple dumps of task-related storage areas 368
levels, application program logical 301
LIBRARY 118
library lookaside (LLA) 128
light pen-detectable field 774
LINE command
  CEBR transaction 195
line length on printing 662
line traffic reduction 778
LINECOUNT option 85
LINK
  migrating to channels and containers 297
LINK command 244, 253, 301
  CHANNEL option 300, 304
  COMMAREA option 300, 302, 304
  IMMEDIATE option 304
  INPUTMSG option 300, 304
  TRANSID option 304
link pack area (LPA) 128
LINK PROGRAM 384
link to program, expecting return 301
link-edit 79, 84
link-edit of map 698
LINKAGE option 85
LIST option 751
LLA (library lookaside) 128
LOAD command
  HOLD option 300
load libraries
  support for secondary extents 134
local copy key 673
locale support in C and C++ 52
locality of reference 255
LOCKED affinity relation 342
logging 261
logical device component 770
logical levels, application program 31, 301
logical messages, rules for 735
logical record presentation 540
logical unit of work (LUW)
  description 224
  recoverable resources 224
  syncpoints used 348
logical units (LUs)
  facilities for 539
lookaside transaction 764
LPA 128
LU type 4
  batch data interchange 546
  device 524
  logical record presentation 540
LUs (logical units)
  facilities for 539

M

magnetic slot reader, 10/63 772
main storage 253
main temporary storage 245
map
  BMS 689
  creating 691
  initializing output 710
  link-edit 698
  moving data to 711
  sets 699
  symbolic input 724
  symbolic output 724
map sets
  Adding a CSECT 157
  Using symbolic description map sets in a program 132
MAPATTS option 711
MAPCOLUMN option 743
MAPFAIL condition 727, 733
MAPHEIGHT option 743
MAPLINE option 743
MAPONLY option 713, 715, 778
MAPPED option 749
mapping input data 725
mapping language structures to XML 455
mapping to C and C++ 420, 422
mapping to COBOL 411, 415
mapping to PL/I 426, 430
mapping XML schemas to application data 458
maps
  BMS 777, 779
  device-dependent 705
  floating 739
  sets 256
MAPSET option 713
MAPSET resource definition 698
mapsets
  loading above the 16MB line 149
MAPWIDTH option 743
MARGINS option 85
MASSINSERT option 514, 517
maxOccurs
  in XML schema 435
MDT 727, 776
MERGE statement, COBOL 24
message routing 750
message title 757
messages, undeliverable 756
migrating START data to use channels 298
migration
  of START data to use channels 298
minOccurs
  in XML schema 435

mixed addressing mode transaction 306
mixing languages 15
modernising COMMAREAs 267
modified data tag 727, 776
modifying execution, EDF 180
modular program 256
MONITOR command 367
MONITOR POINT command 367
monitoring application performance 367
moving data to map 711
MSGINTEG option 549
MSR 772
MSR option 735, 772
multimap screens 779
multipage outputs 779
multiple base registers 65
multithread testing 161
multithreading 230
MVS subspace 591
MVS transaction 306
MXT parameter 258

N

named counters 637
  CICS API 640
  counter name 638
  coupling facility list structure 638
  current value 638
  DFHNC001 638
  DFHNCO macro 638
  maximum value 638
  minimum value 638
  named counter fields 638
  options table 638
  overview 637
  pools 638
native runtime libraries 11
NATLANG option 85
nested programs in COBOL 41
NLEOM option 661, 662, 663, 735
NOAUTOPAGE option 737
NOCBLCARD option 85
NOCPSM option 85
NODE option 677
NODEBUG option 85
NOEDF option 85
NOEDIT option 749
NOEPILOG option 85
NOEPILOG translator option 65
NOFEPI option 85
NOFLUSH option 740, 746
NOHANDLE option 351, 357
NOJBUFSP condition 259
NOLENGTH option 85
NOLINKAGE option 85
non-CICS printer 659
Non-CICS printer 670
non-terminal transactions
  EDF 179
nonconversational programming 223
NONUM option 85
NOOPSEQUENCE option 85
NOOPTIONS option 85
NOPROLOG option 85
NOPROLOG translator option 65
NOQUEUE option 259
NOSEQ option 85
NOSEQUENCE option 85, 94
NOSOURCE option 85
NOSPACE condition 357
NOSPIE option 85
NOSUSPEND option 259
  GETMAIN command 580
  READ command 503
  READNEXT command 513
  READPREV command 513
  WRITE command 514
NOTRUNCATE option 527
NOVBREF option 85
NOWAIT option 546
NOXREF option 85
null parameters, example of DFHNCTR CALLs with 650
null values, use of 779
NUM option 85
NUMREC option 512
NUMSEGMENTS option
  multiple dumps of task-related storage areas 368

O

OCCURS option 701
OO COBOL
  support 23
OPCLASS option 751
OPEN statement, COBOL 24
open transaction environment (OTE) 232
Open Transaction Environment (OTE)
  CONCURRENCY(REQUIRED) programs 238
  OPENAPI programs 239
    restrictions 241
OPENAPI 239
  restrictions 241
operating system waits 259
OPID option 751
OPIDENT value 751
OPMARGINS option 85
OPSEQUENCE option 85
OPSYN (equate operation code) assembler instruction 61
options
  HANDLE CONDITION command 356
  on function keys, EDF 182
OPTIONS option 85
OPTIONS(MAIN) specification 55
OTE, open transaction environment 232
outboard controller 545
outboard formatting 776
output data, chaining of 539
output map, initializing 710
output map, symbolic 724
OVERFLOW condition 740
overlays 256
overtyping EDF displays 180
overview
  dynamic routing with channels 288

P

PA key 665
page break 740
page building operations 779
page fault 254
page overflow 757
page routing operations 779
PAGENUM option 758
PAGESIZE value 739
paging
  reducing effects 254
PAGING option 662, 713, 720
parameters
  null 650
parsing XML 463
partition sets
  installing 158
  loading above the 16MB line 149
partition, active 767
partitioned data set (PDS)
  as document template 613
partitions 763
partitions, defining 765
PARTITIONSET option 766
PARTN option 768
partner, conversation 523
partners, range of 383
PARTNFAIL condition 769
PARTNPAGE option 769
passing control, anticipating return (LINK) 301
passing data to other program 302
pen-detectable field 774
PERFORM command 5
PERFORM DUMP command 367
PF (program function) key 193
physical keys 493
physical map sets
  installing 152
PL/I 55
  fetched procedures 58
  FLOAT compiler option 55
  Language Environment requirements 57
  mapping to XML Schema 426, 430
  mixing with other languages 15
  OPTIONS(MAIN) specification 55
  programming techniques 55, 57, 58
  restrictions 55
platform
  getting started 107
PLT program 584
plus 32 K COMMAREAs 267
plus 32K COMMAREAs 288, 295
policy
  getting started 107
POP HANDLE command 354, 360, 361
POST command 569, 570
pre-translated code 145
preprinted form 601
presentation space 763
PRGDLY option 756
principal facility 226
print control bit 662
print formatting 544
print key 673
print key, Hardware 673

printed output, requests for 660
printer
3270 660
options 661
CICS 659
determining characteristics of 666
non-CICS 659
Non-CICS 670
SCS 663
PRINTERCOMP option 664
printing 659
CICS API considerations 671
line length on 662
START command 668
transient data 668
with BMS routing 670
processing services 476
program
as document template 615
size 253
testing 165
program control
linking to another program 301
passing data to another program 302
program logical levels 301
program design
conversational 223, 549
nonconversational 223
pseudoconversational 223
program labels in EDF 180
PROGRAM option 361
PROGRAM resource definition 698
program storage 245
programming models 223
programming restrictions
assembler language 61
C and C++ 46
COBOL 24, 33, 37
PL/I 55
VS COBOL II 28
programming techniques
assembler language 61, 69, 71
assembly language 71
C and C++ 46
COBOL 24, 29, 33
general 253
PL/I 55, 57, 58
PROLOG option 85
PROTECT option 549
pseudoconversational programming 223
PURGE command, CEBR transaction 196
purge delay 756
PURGE MESSAGE command 736, 750
PUSH HANDLE command 354, 360, 361
PUT command, CEBR transaction 196

Q

QBUSY condition 259
QUERY SECURITY command 368
NOLOG option 369
RESCLASS option 368
RESID option 368
RESTYPE option 368
query transaction 226
querying XML 462
QUEUE command, CEBR transaction 196
queues
extrapartition 598
intrapartition 597
transient data 597
QUOTE option 85
QZERO condition 599

R

RACF 368
range of partners 383
RBA (relative byte address) 481, 485
RDF 517
READ command 510
CONSISTENT option 503
NOSUSPEND option 503
REPEATABLE option 503
UNCOMMITTED option 503
read only containers 283
READ statement, COBOL 24
read-ahead queueing 524
read-only DSA (RDSA) 129
reading data from a display 725
reading from a formatted screen 727
reading records 502
READNEXT command 506
NOSUSPEND option 513
READPREV command 506
NOSUSPEND option 513
READQ TD command 259, 597
READQ TS command 601
ITEM option 601
RECEIVE command 523, 525, 526, 527, 543, 549
MAPFAIL condition 733
RECEIVE MAP command 725
ASIS option 728
RECEIVE PARTN command 768
record
identification 484, 493
locking 488
locking (RLS) 489
record description field 517
record locking 489
record-level sharing (RLS)
accessing files in RLS mode 483
records
adding 514
adding to BDAM data set 516
browsing 502
deleting 512
journal 346
length of 249
reading 502
updating 510
writing 510, 514
recoverable resources 224
exclusive use 224
recovery
of resources 257
problem avoidance 250
sequential terminal support 543
syncpoint 348
reduction of line traffic 778
reentrancy 230
reference modification 37
reference set 256
regression testing 161
relative byte address (RBA) 481, 485
relative byte addressing (RBA)
upgrading 486
relative record data set (RRDS) 481
relative record number (RRN) 481, 485
RELEASE 46
RELEASE command
HOLD option 300
RELEASE option 737
RELTYPE keyword 492
remote transactions, EDF 178
remote-linked programs
DPL 179
EDF 179
REMOTENAME option 374
REMOTESYSTEM option 374
RENT attribute 129
REPEATABLE option
READ command 503
repeated fields 701
repeating content 450
REPLACE statement 37
REQID option 569, 735, 756
request/response unit (RU) 539
REQUIRED 238
RESCLASS option 368
RESETBR command 506
RESID option 368
residence mode (RMODE)
options for CICS applications 126
resources
contention 224
control of 224
controlling sequence of access to 574
exclusive control of 257
exclusive use 224
recoverable 224, 257
RESP and RESP2 options
for interface to JES 676
RESP option 351, 357
deactivating NOHANDLE 356
RESP value 350
RESP2 option 351
RESP2 value 350, 351
restrictions
31-bit addressing 61
64–bit registers 61
assembler language 61
C and C++ 46
COBOL 24, 33, 37
PL/I 55
RESTYPE option 368
RETPAGE condition 721, 747
RETRIEVE command 569, 574
RETURN
migrating to channels and containers 298
RETURN command 244, 304
CHANNEL option 266, 300
COMMAREA option 265, 300
IMMEDIATE option 524, 541
INPUTMSG option 300, 304
TRANSID option 225
reverse interrupt 525

REWRITE command 510
REWRITE statement, COBOL 24
RIDFLD option 492, 506
RMODE (residence mode)
  options for CICS applications 126
RMODE compiler option 24
ROUTE command 750
  LIST option 751
  page overflow 757
  TITLE option 757
route list 751
  LIST option 753
  segment chain entry format 753
  standard entry format 753
ROUTEDMSGS option 751
routing terminals 758
routing, Transaction 369
RPTOPTS 17
RRDS (relative record data set) 481
RRN (relative record number) 481, 485
RTEFAIL condition 753
RTESOME condition 753
RU (request/response unit) 539
rules for logical messages 735
run unit 31
runaway tasks 258
runtime libraries
  Language Environment 11
  native 11
runtime options for Language Environment 17
  STORAGE 24
RUWAPOOL 14
RVI 525

S

SAA Resource Recovery 348
SAA Resource Recovery interface module, DFHCPLRR 98
SAM 521
SCA 469
  best practices 472
  channel-based services 473
  composites 470
  creating SCA composites
    best practices 472
  overview 469
  wiring 470
  XML-based services 475
scenarios
  multiple interactive components 272
  one channel—one program 270
  one channel—several programs 271
  several channels, one component 271
scope of a channel
  example, with LINK commands 279
  example, with XCTL commands 281
  overview 279
screen copy, BMS 673
screen field, 3270 727
screen, reading from a formatted 727
SCS
  printer 663
SCS input 665
SDF II 691, 699
SDF II (IBM Screen Definition Facility II) 149
SEC system initialization option 368
secondary extents, CICS load libraries 134
security
  CICS-defined resource identifiers 368
  EDF 166
  programming hints 368
  record or field level 368
  SEC system initialization option 368
  SPCOMMAND resource type 368
SEGMENTLIST option
  multiple dumps of task-related storage areas 368
selection field 774
SEND command 525, 527, 543, 549
  CNOTCOMPL option 539
  CTLCHAR option 662
  FMH option 541
  INVITE option 523
  LAST option 542
  MSR option 772
SEND CONTROL command 659, 715
SEND MAP command 659, 708
  ACCUM option 735
  ALARM option 714
  CURSOR option 713, 719
  DATAONLY option 713
  ERASE option 714
  ERASEAUP option 714, 746
  FREEKB option 714
  FROM option 713
  LAST option 714
  MAPONLY option 713
  MAPSET option 713
  NOFLUSH option 740, 746
  PAGING option 713, 720
  SET
    command 720
  SET option 713
  TERMINAL option 713, 720
  WAIT option 714
SEND PAGE command 349, 736, 750
  AUTOPAGE option 737
  NOAUTOPAGE option 737
  RELEASE option 737
SEND PARTNSET command 766
SEND TEXT command 659, 745
  MAPPED option 749
  NOEDIT option 749
SEQ option 85
sequence of access to resources, controlling 574
SEQUENCE option 85
sequential terminal support 161, 543
server
  program 374
  region 372, 374
Server Side Include commands 620
service 469
service component architecture
  diagnosing 478
Service Component Architecture
  overview 469
Service-Oriented Architecture 469
SESSBUSY condition 259
SET
  command 5
SET DOCTEMPLATE NEWCOPY command 609
SET option 713, 720
SETLOCALE 46
shared control of records
  VSAM RLS 489
shared data tables 495
SHARED option 243, 249
  GETMAIN command 580
  SHARED option 244
shared storage 249
sharing data across transactions 261
SIGNAL condition 525
simultaneous browse 506
single-screen mode, EDF 176
single-thread testing 161
single-threading 230
size, program 253
SOA 469
SORT statement, COBOL 24
SOURCE option 85
SP option 85
SPACE option 85
space, presentation 763
SPCOMMAND resource type 368
SPIE option 85
SPOLBUSY condition 678
spool
  commands 5
  file 674
SPOOLCLOSE command 674
SPOOLOPEN
  examples 681
SPOOLOPEN command 671, 674
  NODE option 677
  TOKEN option 677
  USERID option 677
SPOOLREAD command 674
SPOOLWRITE command 674
SQL
  references 4
START command 569, 574, 668
START data, migrating to use channels 298
START statement, COBOL 24
STARTBR command 506
Static LIBRARY (DFHRPL) 118
STATIC operand 65
static storage 256
STATREG operand 65
status flag byte, route list 753
STOP statement, COBOL 24
storage
  CICS-key 581
  main 253
  program 245
  shareable 580
  static 256
  user 244
  user-key 581
storage area, dynamic 253
storage control 579
STORAGE option 710
storage protection 579

STORAGE runtime option, Language Environment 24
storage, dynamic 73
struct, C/370 symbolic description map set 151
stubs, EXEC interface 99, 104
subroutines 254
subspace 591
SUSPEND command 573
suspend data set 601
SVC99 46
symbol 606, 619
  defining value 624, 626
  placing in document template 619
symbol list 606, 624, 626
symbol reference 606, 619
symbol substitution 606
symbol table 606
symbolic
  input map 724
  output map 724
symbolic cursor positioning 719
symbolic description map sets
  using in a program 132
symbolic register DFHEIPLR 65
symbols in document templates 605
synchronize action
  journal output 346
SYNCONRETURN option 374, 381
SYNCPOINT command 348, 349, 736
  ROLLBACK option 361
syncpointing 348, 349
syncpointing, DPL 374
SYSEIB option 85
SYSID command, CEBR transaction 196
SYSID option 374
SYSIDERR 371
SYSIN 82
SYSPARM, operand for assembling map sets 151
SYSPRINT 83
SYSPUNCH 83
SYSSTATE macro 65
SYSTEM 46
system information, access to 5
system trace entry point 366

T

tabs, horizontal 665
tabs, vertical 665
task control 573
  sequence of access to resources 574
task-related user exit 584
TASKDATAKEY option 243, 244, 583
TASKDATALOC option 93, 243
TCTUA 265, 583
TCTUAKEY 265, 583
TCTUALOC 265
techniques, programming 253
temporary data 601
temporary storage
  auxiliary 245
  browse transaction, CEBR 191
  data 601
  main 245
temporary storage *(continued)*
  migrating to channels and containers 299
temporary storage queue 245
  as document template 614
  inter-transaction affinity 327
TERM option 706
TERMID value 751
terminal
  contention for 524
  option 713
  sharing 209
  support, sequential 543
  wait 525
TERMINAL
  option 720
TERMINAL command, CEBR transaction 196
terminal control
  bracket protocol, LAST option 541
  break protocol 524
  chaining of input data 539
  chaining of output data 539
  definite response 540
  facilities for logical units 539
  FMH, inbound 541
  FMH, outbound 541
  function management header (FMH) 541
  interrupting 525
  logical record presentation 540
  map input data 725
  print formatting 544
  protocol, break 524
  read-ahead queueing 524
  table user area (TCTUA) 265
Terminal control
  commands 523
  conversation partner 523
  flip-flop mode 523
  half-duplex mode 523
  partner, conversation 523
Terminal control commands 519
TERMINAL option 720
terminals, extended data stream 152
TEST compiler option 24
testing applications
  multithread testing 161
  preparing application table entries 162
  preparing system table entries 162
  preparing the system 162
  regression testing 161
  sequential terminal support 161
  single-thread testing 161
  using sequential devices 161, 543
threadsafe programs 232, 235
  ADDRESS CWA 235
  EXTRACT EXIT 235
  GETMAIN SHARED 235
time field of EIB 5
TIOATDL value 750
TITLE option 757
title, message 757
TOKEN option 511, 677
TOP command, CEBR transaction 196
trace
  description 365
  trace entry point 366
trademarks 786
TRANGRP 338
TRANISO 590
transaction
  affinity 300, 370, 569, 573, 580, 587
  deadlock 500
  routing 369, 370
  routing, dynamic 304
transaction affinity
  inter-transaction affinity 313
  transaction-system affinity 313
transaction groups
  affinity 338
transaction identifier
  CEBR 191
  CECI 205
  CEDF transaction 165
transaction isolation 579
transaction work area 243
transaction-system affinity 313
transactions
  conversational 223
  nonconversational 223
  pseudoconversational 223
TRANSFORM DATATOXML 460
TRANSFORM XMLTODATA 461
transforming binary data to XML 460
transforming XML 463
transforming XML to binary data 461
TRANSID option 304, 374
transient data 668
  extrapartition 248, 260
  intrapartition 248
  queue 248, 330
transient data control
  automatic transaction initiation (ATI) 599
  queues 597, 598
transient data queue
  as document template 614
translation 6, 79
  COBOL 37
translator
  dynamic invocation of 84
  integrated with Language Environment-conforming compilers 79
translator data sets 81, 94
trigger field 773
troubleshooting SCA problems 478
TRUNC compiler option 24
TWA 243
TWASIZE option 243
TYPE=DSECT assembly 698

U

UMT 495
UNCOMMITTED option
  READ command 503
undeliverable messages 756
unescaping 606, 626
unit of work 348
UNTIL option 570

UOW 348
  affinity 342
update operation, BDAM 512
UPDATE option 510
updating records 510
upgrade set 481
uppercase translation in BMS 728
user
  data sets 249
  storage 244
  trace entry point 366
user-key storage 581
user-maintained table 495
user-replaceable module 584
USERDATAKEY option 244, 584
USERID option 677, 679
using channels from JCICS 288
Using dynamic LIBRARY resources 118
utility program
  XML assistant 389

V

validity of reference 255
Variable arrays 435
variables, CECI/CECS 206
VBREF option 85
vertical tabs 665
viewport 763
virtual lookaside facility (VLF) 128
virtual storage 254
virtual storage environment 253
VLF (virtual lookaside facility) 128
VOLUME option 546
VOLUMELENG option 546
VS COBOL II
  Language Environment 28
  Language Environment callable services 12
  programming 28
  support 23
  WORKING-STORAGE limits 24
VSAM
  data sets 517
  enqueues 257
  MASSINSERT option 517
  processor overhead 517

W

WAIT EVENT command 325, 569
WAIT EXTERNAL command 325, 573
WAIT JOURNALNUM command
  synchronize with journal output 346
WAIT option 347, 525, 714
WAIT TERMINAL command 526
wait, terminal 525
WAITCICS command 325, 573
waits, operating system 259
wiring composites 470
WITH DEBUGGING MODE 24
working set 255
WRITE command 514
  NOSUSPEND option 514
WRITE JOURNALNAME command 259, 346
WRITE JOURNALNUM command 259, 346
  create a journal record 346
WRITE statement, COBOL 24
WRITEQ TD command 597
WRITEQ TS command 601
writing records 510, 514
WRKAREA parameter 261
WSDL
  converting to language structure 396
WSDL document
  variable-length 453
  white space 453

X

X8 and X9 TCBs 53
XCTL
  migrating to channels and containers 297
XCTL command 244, 253
  CHANNEL option 300, 304
  COMMAREA option 300, 302
  INPUTMSG option 300, 304
XML
  assistant 389
  data types 463
  parsing 463
  querying 462
  transform to data 389
  transforming 463
  xsd:any data types 464
XML assistant 389
  DFHLS2SC 455
XML binding 455
XML schema 415
  converting to WSDL 390
  variable-length 453
  white space 453
XML Schema 411, 420, 422, 426, 430
XML-based services 475, 476
XOPTS keyword 94
XPCFTCH 54
XPCREQ global user exit 372, 374
XPCTA 54
XPLink 52
  global user exits 54
  non-XPLink objects 53
  TCBs 53
XRBA (extended relative byte address) 485
XREF option 85
xsd:any data types 464
XTC OUT exit 779

Z

z/OS communications Server 521

Readers' Comments — We'd Like to Hear from You

CICS Transaction Server for z/OS
Version 5 Release 1
Application Programming Guide

Publication No. SC34-2844-02

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your 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. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.

Submit your comments using one of these channels:
- Send your comments to the address on the reverse side of this form.
- Send a fax to the following number: +44 1962 816151
- Send your comments via email to: idrcf@uk.ibm.com

If you would like a response from IBM, please fill in the following information:

Name	Address
Company or Organization	
Phone No.	Email address

Readers' Comments — We'd Like to Hear from You
SC34-2844-02

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM United Kingdom Limited
User Technologies Department (MP189)
Hursley Park
Winchester
Hampshire
United Kingdom
SO21 2JN

Fold and Tape | **Please do not staple** | Fold and Tape

SC34-2844-02

Cut or Fold
Along Line

SC34-2844-02